1.   Introduction.

This is METAPOST by John Hobby, a graphics-language processor based on D. E. Knuth's METAFONT.

Much of the original Pascal version of this program was copied with permission from MF.web Version 1.9. It interprets a language very similar to D.E. Knuth's METAFONT, but with changes designed to make it more suitable for PostScript output.

The main purpose of the following program is to explain the algorithms of METAPOST as clearly as possible. However, the program has been written so that it can be tuned to run efficiently in a wide variety of operating environments by making comparatively few changes. Such flexibility is possible because the documentation that follows is written in the WEB language, which is at a higher level than C.

A large piece of software like METAPOST has inherent complexity that cannot be reduced below a certain level of difficulty, although each individual part is fairly simple by itself. The WEB language is intended to make the algorithms as readable as possible, by reflecting the way the individual program pieces fit together and by providing the cross-references that connect different parts. Detailed comments about what is going on, and about why things were done in certain ways, have been liberally sprinkled throughout the program. These comments explain features of the implementation, but they rarely attempt to explain the METAPOST language itself, since the reader is supposed to be familiar with *The METAFONT book* as well as the manual *A User's Manual for MetaPost*, Computing Science Technical Report 162, AT&T Bell Laboratories.

2.   The present implementation is a preliminary version, but the possibilities for new features are limited by the desire to remain as nearly compatible with METAFONT as possible.

On the other hand, the WEB description can be extended without changing the core of the program, and it has been designed so that such extensions are not extremely difficult to make. The *banner* string defined here should be changed whenever METAPOST undergoes any modifications, so that it will be clear which version of METAPOST might be the guilty party when a problem arises.

#define *default_banner*   `"This␣is␣MetaPost,␣Version␣1.999"`
           /∗ printed when METAPOST starts ∗/
#define *true* 1
#define *false* 0

⟨ Metapost version header 2 ⟩ ≡
#define *metapost_version*   `"1.999"`

This code is used in section 3.

3.   The external library header for METAPOST is *mplib.h*. It contains a few typedefs and the header defintions for the externally used fuctions.

The most important of the typedefs is the definition of the structure *MP_options*, that acts as a small, configurable front-end to the fairly large *MP_instance* structure.

⟨ `mplib.h`   3 ⟩ ≡
#ifndef `MPLIB_H`
#define `MPLIB_H` 1
#include `<stdlib.h>`
#ifndef `HAVE_BOOLEAN`
  typedef int boolean;
#endif
  ⟨ Metapost version header 2 ⟩
      typedef struct *MP_instance* ∗**MP**; ⟨ Exported types 15 ⟩
         typedef struct MP_options {
            ⟨ Option variables 26 ⟩
         } **MP_options**; ⟨ Exported function headers 18 ⟩⟨ MPlib header stuff 201 ⟩
#endif

4.   The internal header file is much longer: it not only lists the complete *MP_instance*, but also a lot of functions that have to be available to the PostScript backend, that is defined in a separate `WEB` file.

The variables from **MP_options** are included inside the *MP_instance* wholesale.

⟨ `mpmp.h`   4 ⟩ ≡
#**ifndef** `MPMP_H`
#**define** `MPMP_H`   1
#**include** `"avl.h"`
#**include** `"mplib.h"`
#**include** `<setjmp.h>`
  typedef struct *psout_data_struct* ∗**psout_data**;
  typedef struct *svgout_data_struct* ∗**svgout_data**;
  typedef struct *pngout_data_struct* ∗**pngout_data**;
#**ifndef** `HAVE_BOOLEAN`
  typedef int boolean;
#**endif**
#**ifndef** `INTEGER_TYPE`
  typedef int integer;
#**endif**

  ⟨ Declare helpers 165 ⟩;
  ⟨ Enumeration types 185 ⟩;
  ⟨ Types in the outer block 33 ⟩;
  ⟨ Constants in the outer block 23 ⟩;

  typedef struct MP_instance {
    ⟨ Option variables 26 ⟩⟨ Global variables 14 ⟩
  } **MP_instance**; ⟨ Internal library declarations 10 ⟩⟨ MPlib internal header stuff 6 ⟩
#**endif**

5.
#**define** KPATHSEA_DEBUG_H 1
#**include** <w2c/config.h>
#**include** <stdio.h>
#**include** <stdlib.h>
#**include** <string.h>
#**include** <stdarg.h>
#**include** <assert.h>
#**include** <math.h>
#**ifdef** HAVE_UNISTD_H
#**include** <unistd.h>   /∗ for access ∗/
#**endif**
#**include** <time.h>   /∗ for struct tm **c**o ∗/
#**include** <zlib.h>   /∗ for ZLIB_VERSION, zlibVersion() ∗/
#**include** <png.h>   /∗ for PNG_LIBPNG_VER_STRING, *png_libpng_ver* ∗/
#**include** <pixman.h>   /∗ for PIXMAN_VERSION_STRING, *pixman_version_string*( ) ∗/
#**include** <cairo.h>   /∗ for CAIRO_VERSION_STRING, *cairo_version_string*( ) ∗/
#**include** <gmp.h>   /∗ for *gmp_version* ∗/
#**include** <mpfr.h>   /∗ for MPFR_VERSION_STRING, *mpfr_get_version*( ) ∗/
#**include** "mplib.h"
#**include** "mplibps.h"   /∗ external header ∗/
#**include** "mplibsvg.h"   /∗ external header ∗/
#**include** "mplibpng.h"   /∗ external header ∗/
#**include** "mpmp.h"   /∗ internal header ∗/
#**include** "mppsout.h"   /∗ internal header ∗/
#**include** "mpsvgout.h"   /∗ internal header ∗/
#**include** "mppngout.h"   /∗ internal header ∗/
#**include** "mpmath.h"   /∗ internal header ∗/
#**include** "mpmathdouble.h"   /∗ internal header ∗/
#**include** "mpmathdecimal.h"   /∗ internal header ∗/
#**include** "mpmathbinary.h"   /∗ internal header ∗/
#**include** "mpstrings.h"   /∗ internal header ∗/
    extern *font_number mp_read_font_info*(**MP** *mp*, **char** ∗*fname*);   /∗ tfmin.w ∗/
  ⟨ Preprocessor definitions ⟩

  ⟨ Declarations 8 ⟩;
  ⟨ Basic printing procedures 85 ⟩;
  ⟨ Error handling procedures 112 ⟩

6.   Some debugging support for development. The trick with the variadic macros probably only works in gcc, as this preprocessor feature was not formalized until the c99 standard (and that is too new for us). Lets' hope that at least most compilers understand the non-debug version.

⟨ MPlib internal header stuff 6 ⟩ ≡

#**define** DEBUG   0

#**if** DEBUG

#**define** *debug_number* (*A*)*printf*

   ("%d:␣%s=%.32f␣(%d)\n", __LINE__, #*A*, *number_to_double* (*A*), *number_to_scaled* (*A*))

#**else**

#**define** *debug_number*   (*A*)

#**endif**

#**if** DEBUG > 1

   void *do_debug_printf* (**MP** *mp*, **const char** *∗prefix*, **const char** *∗fmt*, . . .);

#**define** *debug_printf* (*a1*, *a2*, *a3*)*do_debug_printf*   (*mp*, "", *a1*, *a2*, *a3*)

#**define** FUNCTION_TRACE1(*a1*)*do_debug_printf*   (*mp*, "FTRACE:␣", *a1*)

#**define** FUNCTION_TRACE2(*a1*, *a2*)*do_debug_printf*   (*mp*, "FTRACE:␣", *a1*, *a2*)

#**define** FUNCTION_TRACE3(*a1*, *a2*, *a3*)*do_debug_printf*   (*mp*, "FTRACE:␣", *a1*, *a2*, *a3*)

#**define** FUNCTION_TRACE3X(*a1*, *a2*, *a3*)(**void**) *mp*

#**define** FUNCTION_TRACE4(*a1*, *a2*, *a3*, *a4*)*do_debug_printf*   (*mp*, "FTRACE:␣", *a1*, *a2*, *a3*, *a4*)

#**else**

#**define** *debug_printf*   (*a1*, *a2*, *a3*)

#**define** FUNCTION_TRACE1(*a1*)(**void**)   *mp*

#**define** FUNCTION_TRACE2(*a1*, *a2*)(**void**) *mp*

#**define** FUNCTION_TRACE3(*a1*, *a2*, *a3*)(**void**) *mp*

#**define** FUNCTION_TRACE3X(*a1*, *a2*, *a3*)(**void**) *mp*

#**define** FUNCTION_TRACE4(*a1*, *a2*, *a3*, *a4*)(**void**) *mp*

#**endif**

See also sections 36, 67, 82, 174, 193, 235, 251, 262, 267, 270, 273, 455, 458, 462, 469, 473, 477, 482, and 805.

This code is used in section 4.

7.   This function occasionally crashes (if something is written after the log file is already closed), but that is not so important while debugging.

#if DEBUG
  void do_debug_printf (**MP** mp, **const char** $*prefix$, **const char** $*fmt$, ...); **void** do_debug_printf (**MP**
        mp, **const char** $*prefix$, **const char** $*fmt$, ...){ **va_list** ap;
#if 0
      va_start (ap, fmt);
      if (mp→log_file ∧ ¬$ferror$ ((**FILE** $*$) mp→log_file)) {
        $fputs$ ($prefix$, mp→log_file);
        $vfprintf$ (mp→log_file, fmt, ap);
      }
      va_end (ap);
#endif
      va_start (ap, fmt);
#if 0
      if (mp→$term\_out$ ∧ ¬$ferror$ ((**FILE** $*$) mp→$term\_out$)) {
#else
        if ($false$) {
#endif
          $fputs$ ($prefix$, mp→$term\_out$);
          $vfprintf$ (mp→$term\_out$, fmt, ap);
        }
        else {
          $fputs$ ($prefix$, $stdout$);
          $vfprintf$ ($stdout$, fmt, ap);
        }
        va_end (ap);
      }
#endif

8.   Here are the functions that set up the METAPOST instance.

⟨ Declarations 8 ⟩ ≡
  MP_options $*mp\_options$ (**void**);
  MP $mp\_initialize$ (**MP_options** $*opt$);

See also sections 45, 70, 84, 95, 101, 107, 121, 177, 187, 205, 206, 214, 217, 223, 238, 241, 244, 246, 253, 255, 264, 279, 284, 286, 302, 310, 312, 314, 326, 347, 349, 359, 364, 370, 404, 418, 422, 433, 439, 468, 485, 491, 496, 501, 505, 512, 533, 551, 553, 556, 560, 567, 587, 622, 625, 627, 631, 636, 640, 643, 645, 647, 661, 666, 670, 680, 689, 708, 710, 726, 728, 731, 738, 750, 765, 780, 783, 786, 788, 796, 845, 856, 889, 896, 906, 917, 921, 924, 950, 954, 967, 972, 1033, 1036, 1038, 1042, 1044, 1056, 1059, 1088, 1095, 1171, 1234, 1238, 1240, 1272, 1274, 1283, 1288, and 1296.

This code is used in section 5.

9.   **MP_options** $*mp\_options$ (**void**)
  {
    MP_options $*opt$;
    size_t l = **sizeof** (**MP_options**);

    opt = $malloc$ (l);
    if (opt ≠ Λ) {
      $memset$ (opt, 0, l);
    }
    return opt;
  }

10.   ⟨Internal library declarations 10⟩ ≡
  ⟨Declare subroutines for parsing file names 861⟩
See also sections 83, 93, 108, 113, 134, 136, 154, 172, 180, 329, 852, 870, 872, 1096, 1231, 1250, 1253, and 1261.
This code is used in section 4.

11.   The whole instance structure is initialized with zeroes, this greatly reduces the number of statements needed in the *Allocate* ∨ *initialize variables* block.

#define *set_callback_option*(*A*) **do**
        {
          mp→*A* = *mp_*##*A*;
          if (*opt*→*A* ≠ Λ) *mp*→*A* = *opt*→*A*;
        }
        while (0)
  static MP *mp_do_new*(**jmp_buf** *∗buf*)
  {
    MP *mp* = *malloc*(**sizeof**(**MP_instance**));
    if (*mp* ≡ Λ) {
      xfree(*buf*);
      return Λ;
    }
    memset(*mp*, 0, **sizeof**(**MP_instance**));
    mp→*jump_buf* = *buf*;
    return *mp*;
  }

12.   **static void** *mp_free*(**MP** *mp*)
  {
    int *k*;   /∗ loop variable ∗/
    ⟨Dealloc variables 27⟩;
    if (*mp*→*noninteractive*) {
      ⟨Finish non-interactive use 1065⟩;
    }
    xfree(*mp*→*jump_buf*);
    ⟨Free table entries 183⟩;
    free_math();
    xfree(*mp*);
  }

13.   **static void** *mp_do_initialize*(**MP** *mp*)
  {
    ⟨Local variables for initialization 35⟩;
    ⟨Set initial values of key variables 38⟩;
  }

14.   For the retargetable math library, we need to have a pointer, at least.

$\langle$ Global variables 14 $\rangle \equiv$
   void $*math$;

See also sections 25, 29, 37, 47, 60, 65, 73, 76, 77, 105, 109, 111, 138, 142, 150, 166, 175, 181, 194, 208, 210, 216, 225, 291, 325, 340, 345, 367, 384, 430, 447, 543, 545, 604, 605, 610, 614, 623, 634, 667, 674, 679, 685, 691, 719, 730, 762, 766, 807, 822, 841, 844, 865, 893, 899, 926, 929, 986, 999, 1057, 1130, 1141, 1150, 1158, 1167, 1197, 1205, 1211, 1225, 1227, 1247, 1255, 1263, 1279, and 1282.

This code is used in section 4.

15.   ⟨Exported types 15⟩ ≡
    typedef enum {
        $mp\_nan\_type = 0, mp\_scaled\_type, mp\_fraction\_type, mp\_angle\_type, mp\_double\_type, mp\_binary\_type,$
            $mp\_decimal\_type$
    } **mp_number_type**;
    typedef union {
        void *num;
        double $dval$;
        int val;
    } **mp_number_store**;
    typedef struct mp_number_data {
        mp_number_store $data$;
        mp_number_type $type$;
    } **mp_number_data**;
    typedef struct mp_number_data mp_number;
#**define** $is\_number(A)$ $((A).type \neq mp\_nan\_type)$
    typedef void(*$convert\_func$)(**mp_number** *r);
    typedef void(*m_log_func)(**MP** mp, **mp_number** *r, **mp_number** a);
    typedef void(*m_exp_func)(**MP** mp, **mp_number** *r, **mp_number** a);
    typedef void(*$pyth\_add\_func$)(**MP** mp, **mp_number** *r, **mp_number** a, **mp_number** b);
    typedef void(*$pyth\_sub\_func$)(**MP** mp, **mp_number** *r, **mp_number** a, **mp_number** b);
    typedef void(*n_arg_func)(**MP** mp, **mp_number** *r, **mp_number** a, **mp_number** b);
    typedef void(*$velocity\_func$)(**MP** mp, **mp_number** *r, **mp_number** a, **mp_number** b, **mp_number**
        c, **mp_number** d, **mp_number** e);
    typedef void(*$ab\_vs\_cd\_func$)(**MP** mp, **mp_number** *r, **mp_number** a, **mp_number** b, **mp_number**
        c, **mp_number** d);
    typedef void(*$crossing\_point\_func$)(**MP** mp, **mp_number** *r, **mp_number** a, **mp_number**
        b, **mp_number** c);
    typedef void(*$number\_from\_int\_func$)(**mp_number** *A, **int** B);
    typedef void(*$number\_from\_boolean\_func$)(**mp_number** *A, **int** B);
    typedef void(*$number\_from\_scaled\_func$)(**mp_number** *A, **int** B);
    typedef void(*$number\_from\_double\_func$)(**mp_number** *A, **double** B);
    typedef void(*$number\_from\_addition\_func$)(**mp_number** *$A$, **mp_number** B, **mp_number** C);
    typedef void(*$number\_from\_substraction\_func$)(**mp_number** *$A$, **mp_number** B, **mp_number** C);
    typedef void(*$number\_from\_div\_func$)(**mp_number** *A, **mp_number** B, **mp_number** C);
    typedef void(*$number\_from\_mul\_func$)(**mp_number** *A, **mp_number** B, **mp_number** C);
    typedef void(*$number\_from\_int\_div\_func$)(**mp_number** *A, **mp_number** B, **int** C);
    typedef void(*$number\_from\_int\_mul\_func$)(**mp_number** *A, **mp_number** B, **int** C);
    typedef void(*$number\_from\_oftheway\_func$)(**MP** mp, **mp_number** *A, **mp_number** t, **mp_number**
        B, **mp_number** C);
    typedef void(*$number\_negate\_func$)(**mp_number** *A);
    typedef void(*$number\_add\_func$)(**mp_number** *A, **mp_number** B);
    typedef void(*$number\_substract\_func$)(**mp_number** *A, **mp_number** B);
    typedef void(*$number\_modulo\_func$)(**mp_number** *A, **mp_number** B);
    typedef void(*$number\_half\_func$)(**mp_number** *A);
    typedef void(*$number\_halfp\_func$)(**mp_number** *A);
    typedef void(*$number\_double\_func$)(**mp_number** *A);
    typedef void(*$number\_abs\_func$)(**mp_number** *A);
    typedef void(*$number\_clone\_func$)(**mp_number** *A, **mp_number** B);
    typedef void(*$number\_swap\_func$)(**mp_number** *A, **mp_number** *B);
    typedef void(*$number\_add\_scaled\_func$)(**mp_number** *A, **int** b);
    typedef void(*$number\_multiply\_int\_func$)(**mp_number** *A, **int** b);

typedef void(∗*number_divide_int_func*)(**mp_number** ∗*A*, **int** *b*);
typedef int(∗*number_to_int_func*)(**mp_number** *A*);
typedef int(∗*number_to_boolean_func*)(**mp_number** *A*);
typedef int(∗*number_to_scaled_func*)(**mp_number** *A*);
typedef int(∗*number_round_func*)(**mp_number** *A*);
typedef void(∗*number_floor_func*)(**mp_number** ∗*A*);
typedef double(∗*number_to_double_func*)(**mp_number** *A*);
typedef int(∗*number_odd_func*)(**mp_number** *A*);
typedef int(∗*number_equal_func*)(**mp_number** *A*, **mp_number** *B*);
typedef int(∗*number_less_func*)(**mp_number** *A*, **mp_number** *B*);
typedef int(∗*number_greater_func*)(**mp_number** *A*, **mp_number** *B*);
typedef int(∗*number_nonequalabs_func*)(**mp_number** *A*, **mp_number** *B*);
typedef void(∗*make_scaled_func*)(**MP** *mp*, **mp_number** ∗*ret*, **mp_number** *A*, **mp_number** *B*);
typedef void(∗*make_fraction_func*)(**MP** *mp*, **mp_number** ∗*ret*, **mp_number** *A*, **mp_number** *B*);
typedef void(∗*take_fraction_func*)(**MP** *mp*, **mp_number** ∗*ret*, **mp_number** *A*, **mp_number** *B*);
typedef void(∗*take_scaled_func*)(**MP** *mp*, **mp_number** ∗*ret*, **mp_number** *A*, **mp_number** *B*);
typedef void(∗*sin_cos_func*)(**MP** *mp*, **mp_number** *A*, **mp_number** ∗*S*, **mp_number** ∗*C*);
typedef void(∗*slow_add_func*)(**MP** *mp*, **mp_number** ∗*A*, **mp_number** *S*, **mp_number** *C*);
typedef void(∗*sqrt_func*)(**MP** *mp*, **mp_number** ∗*ret*, **mp_number** *A*);
typedef void(∗*init_randoms_func*)(**MP** *mp*, **int** *seed*);
typedef void(∗*new_number_func*)(**MP** *mp*, **mp_number** ∗*A*, **mp_number_type** *t*);
typedef void(∗*free_number_func*)(**MP** *mp*, **mp_number** ∗*n*);
typedef void(∗*fraction_to_round_scaled_func*)(**mp_number** ∗*n*);
typedef void(∗*print_func*)(**MP** *mp*, **mp_number** *A*);
typedef char ∗(∗**tostring_func**)(**MP** *mp*, **mp_number** *A*);
typedef void(∗*scan_func*)(**MP** *mp*, **int** *A*);
typedef void(∗*mp_free_func*)(**MP** *mp*);
typedef void(∗*set_precision_func*)(**MP** *mp*);
typedef struct math_data {
  mp_number *precision_default*;
  mp_number *precision_max*;
  mp_number *precision_min*;
  mp_number *epsilon_t*;
  mp_number *inf_t*;
  mp_number *one_third_inf_t*;
  mp_number *zero_t*;
  mp_number *unity_t*;
  mp_number *two_t*;
  mp_number *three_t*;
  mp_number *half_unit_t*;
  mp_number *three_quarter_unit_t*;
  mp_number *fraction_one_t*;
  mp_number *fraction_half_t*;
  mp_number *fraction_three_t*;
  mp_number *fraction_four_t*;
  mp_number *one_eighty_deg_t*;
  mp_number *three_sixty_deg_t*;
  mp_number *one_k*;
  mp_number *sqrt_8_e_k*;
  mp_number *twelve_ln_2_k*;
  mp_number *coef_bound_k*;
  mp_number *coef_bound_minus_1*;

mp_number *twelvebits_3* ;
mp_number *arc_tol_k* ;
mp_number *twentysixbits_sqrt2_t* ;
mp_number *twentyeightbits_d_t* ;
mp_number *twentysevenbits_sqrt2_d_t* ;
mp_number *fraction_threshold_t* ;
mp_number *half_fraction_threshold_t* ;
mp_number *scaled_threshold_t* ;
mp_number *half_scaled_threshold_t* ;
mp_number *near_zero_angle_t* ;
mp_number *p_over_v_threshold_t* ;
mp_number *equation_threshold_t* ;
mp_number *tfm_warn_threshold_t* ;
mp_number *warning_limit_t* ;

new_number_func allocate ;
free_number_func free ;
number_from_int_func from_int ;
number_from_boolean_func from_boolean ;
number_from_scaled_func from_scaled ;
number_from_double_func from_double ;
number_from_addition_func from_addition ;
number_from_substraction_func from_substraction ;
number_from_div_func from_div ;
number_from_mul_func from_mul ;
number_from_int_div_func from_int_div ;
number_from_int_mul_func from_int_mul ;
number_from_oftheway_func from_oftheway ;
number_negate_func negate ;
number_add_func add ;
number_substract_func substract ;
number_half_func half ;
number_modulo_func modulo ;
number_halfp_func halfp ;
number_double_func do_double ;
number_abs_func abs ;
number_clone_func clone ;
number_swap_func swap ;
number_add_scaled_func add_scaled ;
number_multiply_int_func multiply_int ;
number_divide_int_func divide_int ;
number_to_int_func to_int ;
number_to_boolean_func to_boolean ;
number_to_scaled_func to_scaled ;
number_to_double_func to_double ;
number_odd_func odd ;
number_equal_func equal ;
number_less_func less ;
number_greater_func greater ;
number_nonequalabs_func nonequalabs ;
number_round_func round_unscaled ;
number_floor_func floor_scaled ;
make_scaled_func make_scaled ;

$make\_fraction\_func\ make\_fraction$;
$take\_fraction\_func\ take\_fraction$;
$take\_scaled\_func\ take\_scaled$;
$velocity\_func\ velocity$;
$ab\_vs\_cd\_func\ ab\_vs\_cd$;
$crossing\_point\_func\ crossing\_point$;
$n\_arg\_func\ n\_arg$;
$m\_log\_func\ m\_log$;
$m\_exp\_func\ m\_exp$;
$pyth\_add\_func\ pyth\_add$;
$pyth\_sub\_func\ pyth\_sub$;
$fraction\_to\_round\_scaled\_func\ fraction\_to\_round\_scaled$;
$convert\_func\ fraction\_to\_scaled$;
$convert\_func\ scaled\_to\_fraction$;
$convert\_func\ scaled\_to\_angle$;
$convert\_func\ angle\_to\_scaled$;
$init\_randoms\_func\ init\_randoms$;
$sin\_cos\_func\ sin\_cos$;
$sqrt\_func\ sqrt$;
$slow\_add\_func\ slow\_add$;
$print\_func\ print$;

tostring_func $tostring$;

$scan\_func\ scan\_numeric$;
$scan\_func\ scan\_fractional$;
$mp\_free\_func\ free\_math$;
$set\_precision\_func\ set\_precision$;
} **math_data**;

See also sections 42, 72, 98, 104, 118, 162, 297, 298, 301, 886, 1054, and 1276.

This code is used in section 3.

16.   This procedure gets things started properly.

> **MP** $mp\_initialize(\textbf{MP\_options} * opt)$
> {
>> **MP** mp;
>> **jmp_buf** $*buf = malloc(\textbf{sizeof}(\textbf{jmp\_buf}));$
>>
>> **if** $(buf \equiv \Lambda \lor setjmp(*buf) \neq 0)$ **return** Λ;
>> $mp = mp\_do\_new(buf);$
>> **if** $(mp \equiv \Lambda)$ **return** Λ;
>> $mp \rightarrow userdata = opt \rightarrow userdata;$
>> $mp \rightarrow noninteractive = opt \rightarrow noninteractive;$
>> $set\_callback\_option(find\_file);$
>> $set\_callback\_option(open\_file);$
>> $set\_callback\_option(read\_ascii\_file);$
>> $set\_callback\_option(read\_binary\_file);$
>> $set\_callback\_option(close\_file);$
>> $set\_callback\_option(eof\_file);$
>> $set\_callback\_option(flush\_file);$
>> $set\_callback\_option(write\_ascii\_file);$
>> $set\_callback\_option(write\_binary\_file);$
>> $set\_callback\_option(shipout\_backend);$
>> **if** $(opt \rightarrow banner \land *(opt \rightarrow banner))$ {
>>> $mp \rightarrow banner = xstrdup(opt \rightarrow banner);$
>>
>> }
>> **else** {
>>> $mp \rightarrow banner = xstrdup(default\_banner);$
>>
>> }
>> **if** $(opt \rightarrow command\_line \land *(opt \rightarrow command\_line))$ $mp \rightarrow command\_line = xstrdup(opt \rightarrow command\_line);$
>> **if** $(mp \rightarrow noninteractive)$ {
>>> ⟨ Prepare function pointers for non-interactive use 1061 ⟩;
>>
>> }   /* open the terminal for output */
>> $t\_open\_out();$
> #**if** DEBUG
>> $setvbuf(stdout, (\textbf{char} *) \Lambda, \texttt{\_IONBF}, 0);$
>> $setvbuf(mp \rightarrow term\_out, (\textbf{char} *) \Lambda, \texttt{\_IONBF}, 0);$
> #**endif**
>> **if** $(opt \rightarrow math\_mode \equiv mp\_math\_scaled\_mode)$ {
>>> $mp \rightarrow math = mp\_initialize\_scaled\_math(mp);$
>>
>> }
>> **else if** $(opt \rightarrow math\_mode \equiv mp\_math\_decimal\_mode)$ {
>>> $mp \rightarrow math = mp\_initialize\_decimal\_math(mp);$
>>
>> }
>> **else if** $(opt \rightarrow math\_mode \equiv mp\_math\_binary\_mode)$ {
>>> $mp \rightarrow math = mp\_initialize\_binary\_math(mp);$
>>
>> }
>> **else** {
>>> $mp \rightarrow math = mp\_initialize\_double\_math(mp);$
>>
>> }
>> ⟨ Find and load preload file, if required 854 ⟩;
>> ⟨ Allocate or initialize variables 28 ⟩;
>> $mp\_reallocate\_paths(mp, 1000);$
>> $mp\_reallocate\_fonts(mp, 8);$
>> $mp \rightarrow history = mp\_fatal\_error\_stop;$ /* in case we quit during initialization */

⟨Check the "constant" values for consistency 30⟩;
if (*mp→bad* > 0) {
    char *ss*[256];

    mp_snprintf(*ss*, 256, "Ouch---my␣internal␣constants␣have␣been␣clobbered!\n""---case␣%i",
        (**int**) *mp→bad*);
    mp_fputs((**char** ∗) *ss*, *mp→err_out*);
    ;
    return *mp*;
}
mp_do_initialize(*mp*);   /∗ erase preloaded mem ∗/
mp_init_tab(*mp*);   /∗ initialize the tables ∗/
if (*opt→math_mode* ≡ *mp_math_scaled_mode*) {
    set_internal_string(*mp_number_system*, *mp_intern*(*mp*, "scaled"));
}
else if (*opt→math_mode* ≡ *mp_math_decimal_mode*) {
    set_internal_string(*mp_number_system*, *mp_intern*(*mp*, "decimal"));
}
else if (*opt→math_mode* ≡ *mp_math_binary_mode*) {
    set_internal_string(*mp_number_system*, *mp_intern*(*mp*, "binary"));
}
else {
    set_internal_string(*mp_number_system*, *mp_intern*(*mp*, "double"));
}
mp_init_prim(*mp*);   /∗ call *primitive* for each primitive ∗/
mp_fix_date_and_time(*mp*);
if (¬*mp→noninteractive*) {
    ⟨Initialize the output routines 81⟩;
    ⟨Get the first line of input and prepare to start 1298⟩;
    ⟨Initializations after first line is read 17⟩;
    ⟨Fix up *mp→internal*[*mp_job_name*] 868⟩;
}
else {
    mp→history = *mp_spotless*;
}
set_precision( );
return *mp*;
}

17.   ⟨Initializations after first line is read 17⟩ ≡
mp_open_log_file(*mp*);
mp_set_job_id(*mp*);
mp_init_map_file(*mp*, *mp→troff_mode*);
mp→history = *mp_spotless*;   /∗ ready to go! ∗/
if (*mp→troff_mode*) {
    number_clone(*internal_value*(*mp_gtroffmode*), *unity_t*);
    number_clone(*internal_value*(*mp_prologues*), *unity_t*);
}
if (*mp→start_sym* ≠ Λ) {   /∗ insert the '**everyjob**' symbol ∗/
    set_cur_sym(*mp→start_sym*);
    mp_back_input(*mp*);
}
This code is used in section 16.

18.   ⟨Exported function headers 18⟩ ≡
  extern MP_options *∗mp_options*(**void**);
  extern MP *mp_initialize*(**MP_options** *∗opt*);
  extern int *mp_status*(**MP** *mp*);
  extern void *∗mp_userdata*(**MP** *mp*);
See also sections 116, 133, 197, 377, 379, 1053, 1062, 1070, 1237, and 1293.

This code is used in section 3.

19.   **int** *mp_status*(**MP** *mp*)
  {
    return *mp→history*;
  }

20.   **void** *∗mp_userdata*(**MP** *mp*)
  {
    return *mp→userdata*;
  }

21.   The overall METAPOST program begins with the heading just shown, after which comes a bunch of procedure declarations and function declarations. Finally we will get to the main program, which begins with the comment '*start_here*'. If you want to skip down to the main program now, you can look up '*start_here*' in the index. But the author suggests that the best way to understand this program is to follow pretty much the order of METAPOST's components as they appear in the WEB description you are now reading, since the present ordering is intended to combine the advantages of the "bottom up" and "top down" approaches to the problem of understanding a somewhat complicated system.

22.   Some of the code below is intended to be used only when diagnosing the strange behavior that sometimes occurs when METAPOST is being installed or when system wizards are fooling around with METAPOST without quite knowing what they are doing. Such code will not normally be compiled; it is delimited by the preprocessor test '# **ifdef** DEBUG .. # **endif**'.

23.   The following parameters can be changed at compile time to extend or reduce METAPOST's capacity.
⟨Constants in the outer block 23⟩ ≡
#**define** *bistack_size*   1500
      /∗ size of stack for bisection algorithms; should probably be left at this value ∗/
This code is used in section 4.

24.   Like the preceding parameters, the following quantities can be changed to extend or reduce META-POST's capacity.

25.   ⟨Global variables 14⟩ +≡
  int *pool_size*;   /∗ maximum number of characters in strings, including all error messages and help
        texts, and the names of all identifiers ∗/
  int *max_in_open*;
      /∗ maximum number of input files and error insertions that can be going on simultaneously ∗/
  int *param_size*;   /∗ maximum number of simultaneous macro parameters ∗/

26.   ⟨Option variables 26⟩ ≡

   int *error_line*;   /∗ width of context lines on terminal error messages ∗/

   int *half_error_line*;   /∗ width of first lines of contexts in terminal error messages; should be between
        30 and *error_line* − 15 ∗/

   int *halt_on_error*;   /∗ do we quit at the first error? ∗/

   int *max_print_line*;   /∗ width of longest text lines output; should be at least 60 ∗/

   void ∗*userdata*;   /∗ this allows the calling application to setup local ∗/

   char ∗*banner*;   /∗ the banner that is printed to the screen and log ∗/

   int *ini_version*;

See also sections 43, 48, 50, 66, 99, 119, 151, 163, 195, 853, 866, 887, and 1277.

This code is used in sections 3 and 4.

27.   ⟨Dealloc variables 27⟩ ≡

   xfree(*mp*→*banner*);

See also sections 62, 75, 80, 153, 168, 222, 341, 346, 369, 386, 432, 449, 607, 612, 616, 676, 683, 688, 843, 855, 869, 876, 928, 1064, 1098, 1169, 1199, 1213, 1229, 1257, 1281, and 1290.

This code is used in section 12.

28.

#define   *set_lower_limited_value*(*a*, *b*, *c*)   **do**
       {
         a = *c*;
         if (*b* > *c*) *a* = *b*;
       }
       while (0)

⟨Allocate or initialize variables 28⟩ ≡

   mp→*param_size* = 4;

   mp→*max_in_open* = 0;

   mp→*pool_size* = 10000;

   set_lower_limited_value(*mp*→*error_line*, *opt*→*error_line*, 79);

   set_lower_limited_value(*mp*→*half_error_line*, *opt*→*half_error_line*, 50);

   if (*mp*→*half_error_line* > *mp*→*error_line* − 15) *mp*→*half_error_line* = *mp*→*error_line* − 15;

   mp→*max_print_line* = 100;

   set_lower_limited_value(*mp*→*max_print_line*, *opt*→*max_print_line*, 79);

   mp→*halt_on_error* = (*opt*→*halt_on_error* ? *true* : *false*);

   mp→*ini_version* = (*opt*→*ini_version* ? *true* : *false*);

See also sections 49, 51, 61, 74, 79, 100, 110, 120, 139, 143, 152, 164, 167, 196, 221, 606, 675, 682, 686, 867, 888, 894, 1168, 1228, and 1280.

This code is used in section 16.

29.   In case somebody has inadvertently made bad settings of the "constants," METAPOST checks them using a global variable called *bad*.

   This is the second of many sections of METAPOST where global variables are defined.

⟨Global variables 14⟩ +≡

   integer *bad*;   /∗ is some "constant" wrong? ∗/

30.   Later on we will say ' **if** $(\mathit{int\_packets} + 17 * \mathit{int\_increment} > \mathit{bistack\_size})\ mp{\to}bad = 19;$ ', or something similar.

   In case you are wondering about the non-consequtive values of *bad*: most of the things that used to be WEB constants are now runtime variables with checking at assignment time.

⟨ Check the "constant" values for consistency 30 ⟩ ≡
   $mp{\to}bad = 0;$

See also section 608.

This code is used in section 16.

31.   Here are some macros for common programming idioms.

#define $\mathit{incr}(A)$ $(A) = (A) + 1$ /∗ increase a variable by unity ∗/
#define $\mathit{decr}(A)$ $(A) = (A) - 1$ /∗ decrease a variable by unity ∗/
#define $\mathit{negate}(A)$ $(A) = -(A)$ /∗ change the sign of a variable ∗/
#define   **double**(A) $(A) = (A) + (A)$
#define $\mathit{odd}(A)$ $((A) \% 2 \equiv 1)$

32.   The character set.   In order to make METAPOST readily portable to a wide variety of computers, all of its input text is converted to an internal eight-bit code that includes standard ASCII, the "American Standard Code for Information Interchange." This conversion is done immediately when each character is read in. Conversely, characters are converted from ASCII to the user's external representation just before they are output to a text file.

Such an internal code is relevant to users of METAPOST only with respect to the **char** and **ASCII** operations, and the comparison of strings.

33.   Characters of text that have been converted to METAPOST's internal form are said to be of type *ASCII_code*, which is a subrange of the integers.

⟨Types in the outer block 33⟩ ≡
    typedef unsigned char ASCII_code;   /∗ eight-bit numbers ∗/

See also sections 34, 41, 161, 192, 215, 248, 290, 383, 478, 673, 747, 821, 892, 1058, and 1226.

This code is used in section 4.

34.   The present specification of METAPOST has been written under the assumption that the character set contains at least the letters and symbols associated with ASCII codes 040 through 0176; all of these characters are now available on most computer terminals.

⟨Types in the outer block 33⟩ +≡
    typedef unsigned char text_char;   /∗ the data type of characters in text files ∗/

35.   ⟨Local variables for initialization 35⟩ ≡
    integer *i*;

See also section 149.

This code is used in section 13.

36.   The METAPOST processor converts between ASCII code and the user's external character set by means of arrays *xord* and *xchr* that are analogous to Pascal's *ord* and *chr* functions.

⟨MPlib internal header stuff 6⟩ +≡
#**define** *xchr*(*A*)*mp→xchr* [(*A*)]
#**define** *xord*(*A*)*mp→xord* [(*A*)]

37.   ⟨Global variables 14⟩ +≡
    ASCII_code *xord*[256];   /∗ specifies conversion of input characters ∗/
    text_char *xchr*[256];   /∗ specifies conversion of output characters ∗/

38.   The core system assumes all 8-bit is acceptable. If it is not, a change file has to alter the below section.
    Additionally, people with extended character sets can assign codes arbitrarily, giving an *xchr* equivalent to whatever characters the users of METAPOST are allowed to have in their input files. Appropriate changes to METAPOST's *char_class* table should then be made. (Unlike TeX, each installation of METAPOST has a fixed assignment of category codes, called the *char_class*.) Such changes make portability of programs more difficult, so they should be introduced cautiously if at all.

⟨Set initial values of key variables 38⟩ ≡
    for (*i* = 0; *i* ≤ °*377*; *i*++) {
        xchr(*i*) = (**text_char**) *i*;
    }

See also sections 39, 199, 211, 292, 431, 546, 635, 767, 808, 823, 842, 900, 930, 987, 1142, 1151, 1170, 1232, 1248, and 1256.

This code is used in section 13.

39.   The following system-independent code makes the *xord* array contain a suitable inverse to the information in *xchr*. Note that if $xchr[i] = xchr[j]$ where $i < j < °177$, the value of $xord[xchr[i]]$ will turn out to be j or more; hence, standard ASCII code numbers will be used instead of codes below 040 in case there is a coincidence.

⟨ Set initial values of key variables 38 ⟩ +≡
    for $(i = 0;\ i \le 255;\ i{+}{+})$ {
      $xord(xchr(i)) = °177;$
    }
    for $(i = °200;\ i \le °377;\ i{+}{+})$ {
      $xord(xchr(i)) = (\textbf{ASCII\_code})\ i;$
    }
    for $(i = 0;\ i \le °176;\ i{+}{+})$ {
      $xord(xchr(i)) = (\textbf{ASCII\_code})\ i;$
    }

40.   Input and output.   The bane of portability is the fact that different operating systems treat input and output quite differently, perhaps because computer scientists have not given sufficient attention to this problem. People have felt somehow that input and output are not part of "real" programming. Well, it is true that some kinds of programming are more fun than others. With existing input/output conventions being so diverse and so messy, the only sources of joy in such parts of the code are the rare occasions when one can find a way to make the program a little less bad than it might have been. We have two choices, either to attack I/O now and get it over with, or to postpone I/O until near the end. Neither prospect is very attractive, so let's get it over with.

The basic operations we need to do are (1) inputting and outputting of text, to or from a file or the user's terminal; (2) inputting and outputting of eight-bit bytes, to or from a file; (3) instructing the operating system to initiate ("open") or to terminate ("close") input or output from a specified file; (4) testing whether the end of an input file has been reached; (5) display of bits on the user's screen. The bit-display operation will be discussed in a later section; we shall deal here only with more traditional kinds of I/O.

41.   Finding files happens in a slightly roundabout fashion: the METAPOST instance object contains a field that holds a function pointer that finds a file, and returns its name, or NULL. For this, it receives three parameters: the non-qualified name *fname*, the intended *fopen* operation type *fmode*, and the type of the file *ftype*.

The file types that are passed on in *ftype* can be used to differentiate file searches if a library like kpathsea is used, the fopen mode is passed along for the same reason.

⟨ Types in the outer block 33 ⟩ +≡
    typedef unsigned char eight_bits;   /∗ unsigned one-byte quantity ∗/

42.   ⟨ Exported types 15 ⟩ +≡
  enum mp_filetype {
    mp_filetype_terminal = 0,   /∗ the terminal ∗/
    mp_filetype_error,   /∗ the terminal ∗/
    mp_filetype_program,   /∗ METAPOST language input ∗/
    mp_filetype_log,   /∗ the log file ∗/
    mp_filetype_postscript,   /∗ the postscript output ∗/
    mp_filetype_bitmap,   /∗ the bitmap output file ∗/
    mp_filetype_memfile,   /∗ memory dumps, obsolete ∗/
    mp_filetype_metrics,   /∗ TeX font metric files ∗/
    mp_filetype_fontmap,   /∗ PostScript font mapping files ∗/
    mp_filetype_font,   /∗ PostScript type1 font programs ∗/
    mp_filetype_encoding,   /∗ PostScript font encoding files ∗/
    mp_filetype_text   /∗ first text file for readfrom and writeto primitives ∗/
  };
  typedef char ∗(∗**mp_file_finder**)(**MP**, **const char** ∗, **const char** ∗, **int**);
  typedef void ∗(∗**mp_file_opener**)(**MP**, **const char** ∗, **const char** ∗, **int**);
  typedef char ∗(∗**mp_file_reader**)(**MP**, **void** ∗, **size_t** ∗);
  typedef void(∗*mp_binfile_reader*)(**MP**, **void** ∗, **void** ∗∗, **size_t** ∗);
  typedef void(∗*mp_file_closer*)(**MP**, **void** ∗);
  typedef int(∗*mp_file_eoftest*)(**MP**, **void** ∗);
  typedef void(∗*mp_file_flush*)(**MP**, **void** ∗);
  typedef void(∗*mp_file_writer*)(**MP**, **void** ∗, **const char** ∗);
  typedef void(∗*mp_binfile_writer*)(**MP**, **void** ∗, **void** ∗, **size_t**);

43.   ⟨ Option variables 26 ⟩ +≡
   mp_file_finder *find_file* ;
   mp_file_opener *open_file* ;
   mp_file_reader *read_ascii_file* ;

   mp_binfile_reader read_binary_file ;
   mp_file_closer close_file ;
   mp_file_eoftest eof_file ;
   mp_file_flush flush_file ;
   mp_file_writer write_ascii_file ;
   mp_binfile_writer write_binary_file ;

44.   The default function for finding files is *mp_find_file*. It is pretty stupid: it will only find files in the
current directory.
   static char ∗*mp_find_file*(**MP** *mp* , **const char** ∗*fname* , **const char** ∗*fmode* , **int** *ftype*)
   {
      (**void**) *mp* ;
      if (*fmode*[0] ≠ ᾽r᾽ ∨ (¬*access*(*fname* , R_OK)) ∨ *ftype*) {
         return *mp_strdup*(*fname*);
      }
      return Λ;
   }

45.   Because *mp_find_file* is used so early, it has to be in the helpers section.

⟨ Declarations 8 ⟩ +≡
   static char ∗*mp_find_file*(**MP** *mp* , **const char** ∗*fname* , **const char** ∗*fmode* , **int** *ftype*);
   static void ∗*mp_open_file*(**MP** *mp* , **const char** ∗*fname* , **const char** ∗*fmode* , **int** *ftype*);
   static char ∗*mp_read_ascii_file*(**MP** *mp* , **void** ∗*f* , **size_t** ∗*size*);
   static void *mp_read_binary_file*(**MP** *mp* , **void** ∗*f* , **void** ∗∗*d* , **size_t** ∗*size*);
   static void *mp_close_file*(**MP** *mp* , **void** ∗*f*);
   static int *mp_eof_file*(**MP** *mp* , **void** ∗*f*);
   static void *mp_flush_file*(**MP** *mp* , **void** ∗*f*);
   static void *mp_write_ascii_file*(**MP** *mp* , **void** ∗*f* , **const char** ∗*s*);
   static void *mp_write_binary_file*(**MP** *mp* , **void** ∗*f* , **void** ∗*s* , **size_t** *t*);

46.   The function to open files can now be very short.

```
void *mp_open_file(MP mp, const char *fname, const char *fmode, int ftype)
{
  char realmode[3];

  (void) mp;
  realmode[0] = *fmode;
  realmode[1] = 'b';
  realmode[2] = 0;
  if (ftype ≡ mp_filetype_terminal) {
    return (fmode[0] ≡ 'r' ? stdin : stdout);
  }
  else if (ftype ≡ mp_filetype_error) {
    return stderr;
  }
  else if (fname ≠ Λ ∧ (fmode[0] ≠ 'r' ∨ (¬access(fname, R_OK)))) {
    return (void *) fopen(fname, realmode);
  }
  return Λ;
}
```

47.   (Almost) all file names pass through *name_of_file*.

⟨ Global variables 14 ⟩ +≡
  char *name_of_file*;   /∗ the name of a system file ∗/

48.   If this parameter is true, the terminal and log will report the found file names for input files instead of the requested ones. It is off by default because it creates an extra filename lookup.

⟨ Option variables 26 ⟩ +≡
  int *print_found_names*;   /∗ configuration parameter ∗/

49.   ⟨ Allocate or initialize variables 28 ⟩ +≡
  mp→print_found_names = (*opt→print_found_names* > 0 ? *true* : *false*);

50.   The *file_line_error_style* parameter makes METAPOST use a more standard compiler error message format instead of the Knuthian exclamation mark. It needs the actual version of the current input file name, that will be saved by *open_in* in the *long_name*.

   TODO: currently these long strings cause memory leaks, because they cannot be safely freed as they may appear in the *input_stack* multiple times. In fact, the current implementation is just a quick hack in response to a bug report for metapost 1.205.

#define *long_name*   *mp→cur_input.long_name_field*   /∗ long name of the current file ∗/
⟨ Option variables 26 ⟩ +≡
  int *file_line_error_style*;   /∗ configuration parameter ∗/

51.   ⟨ Allocate or initialize variables 28 ⟩ +≡
  mp→file_line_error_style = (*opt→file_line_error_style* > 0 ? *true* : *false*);

52. METAPOST's file-opening procedures return *false* if no file identified by *name_of_file* could be opened. The *do_open_file* function takes care of the *print_found_names* parameter.

static boolean *mp_do_open_file*(**MP** *mp*, **void** ∗∗*f*, **int** *ftype*, **const char** ∗*mode*)
{
  if (*mp*→*print_found_names* ∨ *mp*→*file_line_error_style*) {
    char ∗*s* = (*mp*→*find_file*)(*mp*, *mp*→*name_of_file*, *mode*, *ftype*);
    if (*s* ≠ Λ) {
      ∗*f* = (*mp*→*open_file*)(*mp*, *mp*→*name_of_file*, *mode*, *ftype*);
      if (*mp*→*print_found_names*) {
        xfree(*mp*→*name_of_file*);
        mp→*name_of_file* = *xstrdup*(*s*);
      }
      if ((∗*mode* ≡ '**r**') ∧ (*ftype* ≡ *mp_filetype_program*)) {
        long_name = *xstrdup*(*s*);
      }
      xfree(*s*);
    }
    else {
      ∗*f* = Λ;
    }
  }
  else {
    ∗*f* = (*mp*→*open_file*)(*mp*, *mp*→*name_of_file*, *mode*, *ftype*);
  }
  return (∗*f* ? *true* : *false*);
}

static boolean *mp_open_in*(**MP** *mp*, **void** ∗∗*f*, **int** *ftype*)
{   /∗ open a file for input ∗/
  return *mp_do_open_file*(*mp*, *f*, *ftype*, "**r**");
}

static boolean *mp_open_out*(**MP** *mp*, **void** ∗∗*f*, **int** *ftype*)
{   /∗ open a file for output ∗/
  return *mp_do_open_file*(*mp*, *f*, *ftype*, "**w**");
}

53.   **static char** $*mp\_read\_ascii\_file$(**MP** mp, **void** $*ff$, **size_t** $*size$)
  {
    int c;
    size_t $len = 0$, $lim = 128$;
    char $*s = \Lambda$;
    FILE $*f = ($**FILE** $*) ff$;
    $*size = 0$;
    (**void**) mp;   /* for -Wunused */
    if $(f \equiv \Lambda)$ **return** Λ;
    $c = fgetc(f)$;
    if $(c \equiv \texttt{EOF})$ **return** Λ;
    $s = malloc(lim)$;
    if $(s \equiv \Lambda)$ **return** Λ;
    while $(c \neq \texttt{EOF} \wedge c \neq \texttt{'\textbackslash n'} \wedge c \neq \texttt{'\textbackslash r'})$ {
      if $((len + 1) \equiv lim)$ {
        $s = realloc(s, (lim + (lim \gg 2)))$;
        if $(s \equiv \Lambda)$ **return** Λ;
        $lim \mathrel{+}= (lim \gg 2)$;
      }
      $s[len \mathbin{++}] = ($**char**$) c$;
      $c = fgetc(f)$;
    }
    if $(c \equiv \texttt{'\textbackslash r'})$ {
      $c = fgetc(f)$;
      if $(c \neq \texttt{EOF} \wedge c \neq \texttt{'\textbackslash n'})$ $ungetc(c, f)$;
    }
    $s[len] = 0$;
    $*size = len$;
    return s;
  }

54.   **void** $mp\_write\_ascii\_file$(**MP** mp, **void** $*f$, **const char** $*s$)
  {
    (**void**) mp;
    if $(f \neq \Lambda)$ {
      $fputs(s, ($**FILE** $*) f)$;
    }
  }

55.   **void** $mp\_read\_binary\_file$(**MP** mp, **void** $*f$, **void** $**data$, **size_t** $*size$)
  {
    size_t $len = 0$;
    (**void**) mp;
    if $(f \neq \Lambda)$ $len = fread(*data, 1, *size, ($**FILE** $*) f)$;
    $*size = len$;
  }

56.   **void** $mp\_write\_binary\_file$(**MP** mp, **void** $*f$, **void** $*s$, **size_t** $size$)
  {
    (**void**) mp;
    if $(f \neq \Lambda)$ (**void**) $fwrite(s, size, 1, ($**FILE** $*) f)$;
  }

57.   **void** *mp_close_file*(**MP** *mp*, **void** *∗f*)
{

   (**void**) *mp*;
   if (*f ≠ Λ*) *fclose*((**FILE** *∗*) *f*);
}

58.   **int** *mp_eof_file*(**MP** *mp*, **void** *∗f*)
{

   (**void**) *mp*;
   if (*f ≠ Λ*) **return** *feof*((**FILE** *∗*) *f*);
   else return 1;
}

59.   **void** *mp_flush_file*(**MP** *mp*, **void** *∗f*)
{

   (**void**) *mp*;
   if (*f ≠ Λ*) *fflush*((**FILE** *∗*) *f*);
}

60.   Input from text files is read one line at a time, using a routine called *input_ln*. This function is defined
in terms of global variables called *buffer*, *first*, and *last* that will be described in detail later; for now, it
suffices for us to know that *buffer* is an array of **ASCII_code** values, and that *first* and *last* are indices
into this array representing the beginning and ending of a line of text.

⟨ Global variables 14 ⟩ +≡
   size_t *buf_size*;
     /∗ maximum number of characters simultaneously present in current lines of open files ∗/
   ASCII_code *∗buffer*;   /∗ lines of characters being read ∗/
   size_t *first*;   /∗ the first unused position in *buffer* ∗/
   size_t *last*;   /∗ end of the line just input to *buffer* ∗/
   size_t *max_buf_stack*;   /∗ largest index used in *buffer* ∗/

61.   ⟨ Allocate or initialize variables 28 ⟩ +≡
   mp→buf_size = 200;
   mp→buffer = *xmalloc*((*mp→buf_size* + 1), **sizeof**(**ASCII_code**));

62.   ⟨ Dealloc variables 27 ⟩ +≡
   xfree(*mp→buffer*);

63.   **static void** *mp_reallocate_buffer*(**MP** *mp*, **size_t** *l*)
{

   ASCII_code *∗buffer*;

   if (*l > max_halfword*) {
     mp_confusion(*mp*, "buffer␣size");   /∗ can't happen (I hope) ∗/
   }
   buffer = *xmalloc*((*l* + 1), **sizeof**(**ASCII_code**));
   (**void**) *memcpy*(*buffer*, *mp→buffer*, (*mp→buf_size* + 1));
   xfree(*mp→buffer*);
   mp→buffer = *buffer*;
   mp→buf_size = *l*;
}

64.   The *input_ln* function brings the next line of input from the specified field into available positions of the buffer array and returns the value *true*, unless the file has already been entirely read, in which case it returns *false* and sets *last*: = *first*. In general, the **ASCII_code** numbers that represent the next line of the file are input into *buffer*[*first*], *buffer*[*first* + 1], ..., *buffer*[*last* − 1]; and the global variable *last* is set equal to *first* plus the length of the line. Trailing blanks are removed from the line; thus, either *last* = *first* (in which case the line was entirely blank) or *buffer*[*last* − 1] <> "␣".

The variable *max_buf_stack*, which is used to keep track of how large the *buf_size* parameter must be to accommodate the present job, is also kept up to date by *input_ln*.

```
static boolean mp_input_ln(MP mp, void ∗f)
{       /∗ inputs the next line or returns false ∗/
   char ∗s;
   size_t size = 0;

   mp→last = mp→first;        /∗ cf. Matthew 19:30 ∗/
   s = (mp→read_ascii_file)(mp, f, &size);
   if (s ≡ Λ) return false;
   if (size > 0) {
      mp→last = mp→first + size;
      if (mp→last ≥ mp→max_buf_stack) {
         mp→max_buf_stack = mp→last + 1;
         while (mp→max_buf_stack > mp→buf_size) {
            mp_reallocate_buffer(mp, (mp→buf_size + (mp→buf_size ≫ 2)));
         }
      }
      (void) memcpy((mp→buffer + mp→first), s, size);
   }
   free(s);
   return true;
}
```

65.   The user's terminal acts essentially like other files of text, except that it is used both for input and for output. When the terminal is considered an input file, the file variable is called *term_in*, and when it is considered an output file the file variable is *term_out*.

⟨ Global variables 14 ⟩ +≡
```
   void ∗term_in;       /∗ the terminal as an input file ∗/
   void ∗term_out;       /∗ the terminal as an output file ∗/
   void ∗err_out;       /∗ the terminal as an output file ∗/
```

66.   Here is how to open the terminal files. In the default configuration, nothing happens except that the command line (if there is one) is copied to the input buffer. The variable *command_line* will be filled by the *main* procedure.

#define *t_open_out*() **do**
    {   /∗ open the terminal for text output ∗/
      mp⃗*term_out* = (*mp*⃗*open_file*)(*mp*, "terminal", "w", *mp_filetype_terminal*);
      mp⃗*err_out* = (*mp*⃗*open_file*)(*mp*, "error", "w", *mp_filetype_error*);
    }
    while (0)
#define *t_open_in*() **do**
    {   /∗ open the terminal for text input ∗/
      mp⃗*term_in* = (*mp*⃗*open_file*)(*mp*, "terminal", "r", *mp_filetype_terminal*);
      if (*mp*⃗*command_line* ≠ Λ) {
        mp⃗*last* = *strlen*(*mp*⃗*command_line*);
        (**void**) *memcpy*((**void** ∗) *mp*⃗*buffer*, (**void** ∗) *mp*⃗*command_line*, *mp*⃗*last*);
        xfree(*mp*⃗*command_line*);
      }
      else {
        mp⃗*last* = 0;
      }
    }
    while (0)
⟨ Option variables 26 ⟩ +≡
  char ∗*command_line*;

67.   Sometimes it is necessary to synchronize the input/output mixture that happens on the user's terminal, and three system-dependent procedures are used for this purpose. The first of these, *update_terminal*, is called when we want to make sure that everything we have output to the terminal so far has actually left the computer's internal buffers and been sent. The second, *clear_terminal*, is called when we wish to cancel any input that the user may have typed ahead (since we are about to issue an unexpected error message). The third, *wake_up_terminal*, is supposed to revive the terminal if the user has disabled it by some instruction to the operating system. The following macros show how these operations can be specified:

⟨ MPlib internal header stuff 6 ⟩ +≡
#define *update_terminal*()(*mp*⃗*flush_file*) (*mp*, *mp*⃗*term_out*)   /∗ empty the terminal output buffer ∗/
#define *clear_terminal* ()   /∗ clear the terminal input buffer ∗/
#define *wake_up_terminal*()(*mp*⃗*flush_file*) (*mp*, *mp*⃗*term_out*)
    /∗ cancel the user's cancellation of output ∗/

68.   We need a special routine to read the first line of METAPOST input from the user's terminal. This line is different because it is read before we have opened the transcript file; there is sort of a "chicken and egg" problem here. If the user types 'input cmr10' on the first line, or if some macro invoked by that line does such an input, the transcript file will be named 'cmr10.log'; but if no input commands are performed during the first line of terminal input, the transcript file will acquire its default name 'mpout.log'. (The transcript file will not contain error messages generated by the first line before the first input command.)

    The first line is even more special. It's nice to let the user start running a METAPOST job by typing a command line like 'MP cmr10'; in such a case, METAPOST will operate as if the first line of input were 'cmr10', i.e., the first line will consist of the remainder of the command line, after the part that invoked METAPOST.

69.   Different systems have different ways to get started. But regardless of what conventions are adopted, the routine that initializes the terminal should satisfy the following specifications:

1) It should open file *term_in* for input from the terminal. (The file *term_out* will already be open for output to the terminal.)
2) If the user has given a command line, this line should be considered the first line of terminal input. Otherwise the user should be prompted with '**', and the first line of input should be whatever is typed in response.
3) The first line of input, which might or might not be a command line, should appear in locations *first* to *last* − 1 of the *buffer* array.
4) The global variable *loc* should be set so that the character to be read next by METAPOST is in *buffer*[*loc*]. This character should not be blank, and we should have *loc* < *last*.

(It may be necessary to prompt the user several times before a non-blank line comes in. The prompt is '**' instead of the later '*' because the meaning is slightly different: '`input`' need not be typed immediately after '**'.)

#define   *loc*   *mp*→*cur_input.loc_field*   /∗ location of first unread character in *buffer* ∗/

   boolean *mp_init_terminal*(**MP** *mp*)
  {   /∗ gets the terminal input started ∗/
    t_open_in( );
    if (*mp*→*last* ≠ 0) {
      loc = 0;
      mp→*first* = 0;
      return *true*;
    }
    while (1) {
      if (¬*mp*→*noninteractive*) {
        wake_up_terminal( );
        mp_fputs("**", *mp*→*term_out*);
        ;
        update_terminal( );
      }
      if (¬*mp_input_ln*(*mp*, *mp*→*term_in*)) {   /∗ this shouldn't happen ∗/
        mp_fputs("\n!␣End␣of␣file␣on␣the␣terminal...␣why?", *mp*→*term_out*);
        ;
        return *false*;
      }
      loc = (*halfword*)*mp*→*first*;
      while ((*loc* < (**int**) *mp*→*last*) ∧ (*mp*→*buffer*[*loc*] ≡ '␣')) *incr*(*loc*);
      if (*loc* < (**int**) *mp*→*last*) {
        return *true*;   /∗ return unless the line was all blank ∗/
      }
      if (¬*mp*→*noninteractive*) {
        mp_fputs("Please␣type␣the␣name␣of␣your␣input␣file.\n", *mp*→*term_out*);
      }
    }
  }

70.   ⟨ Declarations 8 ⟩ +≡
  static boolean *mp_init_terminal*(**MP** *mp*);

71. Globals for strings.

72.   Symbolic token names and diagnostic messages are variable-length strings of eight-bit characters. Many strings METAPOST uses are simply literals in the compiled source, like the error messages and the names of the internal parameters. Other strings are used or defined from the METAPOST input language, and these have to be interned.

METAPOST uses strings more extensively than METAFONT does, but the necessary operations can still be handled with a fairly simple data structure. The avl tree *strings* contains all of the known string structures.

Each structure contains an **unsigned char** pointer containing the eight-bit data, a **size_t** that holds the length of that data, and an **int** that indicates how often this string is referenced (this will be explained below). Such strings are referred to by structure pointers called *mp_string*.

Besides the avl tree, there is a set of three variables called *cur_string*, *cur_length* and *cur_string_size* that are used for strings while they are being built.

⟨ Exported types 15 ⟩ +≡
> **typedef struct** {
>   **unsigned char** *∗str*;   /∗ the string value ∗/
>   **size_t** *len*;   /∗ its length ∗/
>   **int** *refs*;   /∗ number of references ∗/
> } **mp_lstring**;
> **typedef mp_lstring ∗mp_string**;   /∗ for pointers to string values ∗/

73.   The string handling functions are in `mpstrings.w`, but strings need a bunch of globals and those are defined here in the main file.

⟨ Global variables 14 ⟩ +≡
> *avl_tree strings*;   /∗ string avl tree ∗/
>
> **unsigned char** *∗cur_string*;   /∗ current string buffer ∗/
> **size_t** *cur_length*;   /∗ current index in that buffer ∗/
> **size_t** *cur_string_size*;   /∗ malloced size of *cur_string* ∗/

74.   ⟨ Allocate or initialize variables 28 ⟩ +≡
> *mp_initialize_strings*(*mp*);

75.   ⟨ Dealloc variables 27 ⟩ +≡
> *mp_dealloc_strings*(*mp*);

76.   The next four variables are for keeping track of string memory usage.

⟨ Global variables 14 ⟩ +≡
> **integer** *pool_in_use*;   /∗ total number of string bytes actually in use ∗/
> **integer** *max_pl_used*;   /∗ maximum *pool_in_use* so far ∗/
> **integer** *strs_in_use*;   /∗ total number of strings actually in use ∗/
> **integer** *max_strs_used*;   /∗ maximum *strs_in_use* so far ∗/

77.   On-line and off-line printing.   Messages that are sent to a user's terminal and to the transcript-log file are produced by several '*print*' procedures. These procedures will direct their output to a variety of places, based on the setting of the global variable *selector*, which has the following possible values:

term_and_log, the normal setting, prints on the terminal and on the transcript file.
log_only, prints only on the transcript file.
term_only, prints only on the terminal.
no_print, doesn't print at all. This is used only in rare cases before the transcript file is open.
pseudo, puts output into a cyclic buffer that is used by the *show_context* routine; when we get to that routine
      we shall discuss the reasoning behind this curious mode.
new_string, appends the output to the current string in the string pool.
$\geq$ *write_file* prints on one of the files used for the **write** command.

The symbolic names '*term_and_log*', etc., have been assigned numeric codes that satisfy the convenient relations $no\_print + 1 = term\_only$, $no\_print + 2 = log\_only$, $term\_only + 2 = log\_only + 1 = term\_and\_log$. These relations are not used when *selector* could be *pseudo*, or *new_string*. We need not check for unprintable characters when $selector < pseudo$.

   Three additional global variables, *tally*, *term_offset* and *file_offset* record the number of characters that have been printed since they were most recently cleared to zero. We use *tally* to record the length of (possibly very long) stretches of printing; *term_offset*, and *file_offset*, on the other hand, keep track of how many characters have appeared so far on the current line that has been output to the terminal, the transcript file, or the PostScript output file, respectively.

#define   *new_string*   0   /∗ printing is deflected to the string pool ∗/
#define   *pseudo*   2   /∗ special *selector* setting for *show_context* ∗/
#define   *no_print*   3   /∗ *selector* setting that makes data disappear ∗/
#define   *term_only*   4   /∗ printing is destined for the terminal only ∗/
#define   *log_only*   5   /∗ printing is destined for the transcript file only ∗/
#define   *term_and_log*   6   /∗ normal *selector* setting ∗/
#define   *write_file*   7   /∗ first write file selector ∗/

⟨ Global variables 14 ⟩ +≡
  void ∗*log_file*;   /∗ transcript of METAPOST session ∗/
  void ∗*output_file*;   /∗ the generic font output goes here ∗/
  unsigned int *selector*;   /∗ where to print a message ∗/
  integer *tally*;   /∗ the number of characters recently printed ∗/
  unsigned int *term_offset*;   /∗ the number of characters on the current terminal line ∗/
  unsigned int *file_offset*;   /∗ the number of characters on the current file line ∗/
  ASCII_code ∗*trick_buf*;   /∗ circular buffer for pseudoprinting ∗/
  integer *trick_count*;   /∗ threshold for pseudoprinting, explained later ∗/
  integer *first_count*;   /∗ another variable for pseudoprinting ∗/

78.   The first 128 strings will contain 95 standard ASCII characters, and the other 33 characters will be printed in three-symbol form like '`^^A`' unless a system-dependent change is made here. Installations that have an extended character set, where for example $xchr[°32] = $ '≠', would like string 032 to be printed as the single character 032 instead of the three characters 0136, 0136, 0132 (`^^Z`). On the other hand, even people with an extended character set will want to represent string 015 by `^^M`, since 015 is ASCII's "carriage return" code; the idea is to produce visible strings instead of tabs or line-feeds or carriage-returns or bell-rings or characters that are treated anomalously in text files.

The boolean expression defined here should be *true* unless METAPOST internal code number k corresponds to a non-troublesome visible symbol in the local character set. If character k cannot be printed, and $k < °200$, then character $k + °100$ or $k - °100$ must be printable; moreover, ASCII codes $[°60..071, °141..0146]$ must be printable.

⟨ Character k cannot be printed 78 ⟩ ≡
    $(k < $ '␣' $) \vee (k \equiv 127)$

This code is used in section 87.

79.   ⟨ Allocate or initialize variables 28 ⟩ +≡
    $mp\text{-}trick\_buf = xmalloc((mp\text{-}error\_line + 1), \textbf{sizeof}(\textbf{ASCII\_code}));$

80.   ⟨ Dealloc variables 27 ⟩ +≡
    $xfree(mp\text{-}trick\_buf);$

81.   ⟨ Initialize the output routines 81 ⟩ ≡
    $mp\text{-}selector = term\_only;$
    $mp\text{-}tally = 0;$
    $mp\text{-}term\_offset = 0;$
    $mp\text{-}file\_offset = 0;$

See also section 90.

This code is used in sections 16 and 1066.

82.   Macro abbreviations for output to the terminal and to the log file are defined here for convenience. Some systems need special conventions for terminal output, and it is possible to adhere to those conventions by changing *wterm*, *wterm_ln*, and *wterm_cr* here.

⟨ MPlib internal header stuff 6 ⟩ +≡
#define *mp_fputs*(*b*, *f*)(*mp*→*write_ascii_file*) (*mp*, *f*, *b*)
#define *wterm*(*A*)*mp_fputs* ((*A*), *mp*→*term_out*)
#define *wterm_chr*(*A*)
  {
    unsigned char *ss*[2];
    ss[0] = (*A*);
    ss[1] = '\0';
    wterm((**char** ∗) *ss*);
  }
#define *wterm_cr mp_fputs* ("\n", *mp*→*term_out*)
#define *wterm_ln*(*A*)
  {
    wterm_cr;
    mp_fputs((*A*), *mp*→*term_out*);
  }
#define *wlog*(*A*)*mp_fputs* ((*A*), *mp*→*log_file*)
#define *wlog_chr*(*A*)
  {
    unsigned char *ss*[2];
    ss[0] = (*A*);
    ss[1] = '\0';
    wlog((**char** ∗) *ss*);
  }
#define *wlog_cr mp_fputs* ("\n", *mp*→*log_file*)
#define *wlog_ln*(*A*)
  {
    wlog_cr;
    mp_fputs((*A*), *mp*→*log_file*);
  }

83.   To end a line of text output, we call *print_ln*. Cases 0..*max_write_files* use an array *wr_file* that will be declared later.

#define *mp_print_text*(*A*) *mp_print_str*(*mp*, *text*((*A*)))

⟨ Internal library declarations 10 ⟩ +≡
  void *mp_print*(**MP** *mp*, **const char** ∗*s*);
  void *mp_printf*(**MP** *mp*, **const char** ∗*ss*, . . .);
  void *mp_print_ln*(**MP** *mp*);
  void *mp_print_char*(**MP** *mp*, **ASCII_code** *k*);
  void *mp_print_str*(**MP** *mp*, **mp_string** *s*);
  void *mp_print_nl*(**MP** *mp*, **const char** ∗*s*);
  void *mp_print_two*(**MP** *mp*, **mp_number** *x*, **mp_number** *y*);

84.   ⟨ Declarations 8 ⟩ +≡
  static void *mp_print_visible_char*(**MP** *mp*, **ASCII_code** *s*);

85.   ⟨Basic printing procedures 85⟩ ≡
  void *mp_print_ln*(**MP** *mp*)
  {   /∗ prints an end-of-line ∗/
    switch (*mp*→*selector*) {
    case *term_and_log*: *wterm_cr*;
      wlog_cr;
      mp→*term_offset* = 0;
      mp→*file_offset* = 0;
      break;
    case *log_only*: *wlog_cr*;
      mp→*file_offset* = 0;
      break;
    case *term_only*: *wterm_cr*;
      mp→*term_offset* = 0;
      break;
    case *no_print*: **case** *pseudo*: **case** *new_string*: **break**;
    default: *mp_fputs*("\n", *mp*→*wr_file*[(*mp*→*selector* − *write_file*)]);
    }
  }   /∗ note that *tally* is not affected ∗/

See also sections 86, 87, 88, 89, 91, 92, 147, 188, 207, 209, and 849.

This code is used in section 5.

86.   The *print_visible_char* procedure sends one character to the desired destination, using the *xchr* array
to map it into an external character compatible with *input_ln*. (It assumes that it is always called with
a visible ASCII character.)   All printing comes through *print_ln* or *print_char*, which ultimately calls
print_visible_char, hence these routines are the ones that limit lines to at most *max_print_line* characters.
But we must make an exception for the PostScript output file since it is not safe to cut up lines arbitrarily
in PostScript.

⟨ Basic printing procedures 85 ⟩ +≡
```
    static void mp_print_visible_char(MP mp, ASCII_code s)
    {     /* prints a single character */
      switch (mp→selector) {
      case term_and_log: wterm_chr(xchr(s));
        wlog_chr(xchr(s));
        incr(mp→term_offset);
        incr(mp→file_offset);
        if (mp→term_offset ≡ (unsigned) mp→max_print_line) {
          wterm_cr;
          mp→term_offset = 0;
        }
        ;
        if (mp→file_offset ≡ (unsigned) mp→max_print_line) {
          wlog_cr;
          mp→file_offset = 0;
        }
        ;
        break;
      case log_only: wlog_chr(xchr(s));
        incr(mp→file_offset);
        if (mp→file_offset ≡ (unsigned) mp→max_print_line)  mp_print_ln(mp);
        break;
      case term_only: wterm_chr(xchr(s));
        incr(mp→term_offset);
        if (mp→term_offset ≡ (unsigned) mp→max_print_line)  mp_print_ln(mp);
        break;
      case no_print: break;
      case pseudo:
        if (mp→tally < mp→trick_count)  mp→trick_buf[mp→tally % mp→error_line] = s;
        break;
      case new_string: append_char(s);
        break;
      default:
        {
          text_char ss[2] = {0, 0};
          ss[0] = xchr(s);
          mp_fputs((char *) ss, mp→wr_file[(mp→selector − write_file)]);
        }
      }
      incr(mp→tally);
    }
```

87.   The *print_char* procedure sends one character to the desired destination. File names and string expressions might contain **ASCII_code** values that can't be printed using *print_visible_char*. These characters will be printed in three- or four-symbol form like '`^^A`' or '`^^e4`'. (This procedure assumes that it is safe to bypass all checks for unprintable characters when *selector* is in the range $0 .. max\_write\_files - 1$. The user might want to write unprintable characters.

⟨ Basic printing procedures 85 ⟩ +≡
```
  void mp_print_char(MP mp, ASCII_code k)
{      /* prints a single character */
  if (mp→selector < pseudo ∨ mp→selector ≥ write_file) {
    mp_print_visible_char(mp, k);
  }
  else if (⟨ Character k cannot be printed 78 ⟩) {
    mp_print(mp, "^^");
    if (k < °100) {
      mp_print_visible_char(mp, (ASCII_code)(k + °100));
    }
    else if (k < °200) {
      mp_print_visible_char(mp, (ASCII_code)(k - °100));
    }
    else {
      int l;      /* small index or counter */
      l = (k/16);
      mp_print_visible_char(mp, xord(l < 10 ? l + '0' : l - 10 + 'a'));
      l = (k % 16);
      mp_print_visible_char(mp, xord(l < 10 ? l + '0' : l - 10 + 'a'));
    }
  }
  else {
    mp_print_visible_char(mp, k);
  }
}
```

88.   An entire string is output by calling *print*. Note that if we are outputting the single standard ASCII character c, we could call *print*(`"c"`), since `"c"` = 99 is the number of a single-character string, as explained above. But *print_char*(`"c"`) is quicker, so METAPOST goes directly to the *print_char* routine when it knows that this is safe. (The present implementation assumes that it is always safe to print a visible ASCII character.)

⟨Basic printing procedures 85⟩ +≡
  static void *mp_do_print*(**MP** *mp*, **const char** *∗ss*, **size_t** *len*)
  {   /∗ prints string *s* ∗/
    if (*len* ≡ 0) **return**;
    if (*mp*→*selector* ≡ *new_string*) {
      str_room(*len*);
      memcpy((*mp*→*cur_string* + *mp*→*cur_length*), *ss*, *len*);
      mp→*cur_length* += *len*;
    }
    else {
      size_t *j* = 0;
      while (*j* < *len*) {   /∗ this was *xord*((**int**) *ss*[*j*]) but that doesnt work ∗/
        mp_print_char(*mp*, (**ASCII_code**) *ss*[*j*]);
        j++;
      }
    }
  }

89.

⟨Basic printing procedures 85⟩ +≡
  void *mp_print*(**MP** *mp*, **const char** *∗ss*)
  {
    assert(*ss* ≠ Λ);
    mp_do_print(*mp*, *ss*, *strlen*(*ss*));
  }
  void *mp_printf*(**MP** *mp*, **const char** *∗ss*, ...)
  {
    va_list *ap*;
    char *pval*[256];
    assert(*ss* ≠ Λ);
    va_start(*ap*, *ss*);
    vsnprintf(*pval*, 256, *ss*, *ap*);
    mp_do_print(*mp*, *pval*, *strlen*(*pval*));
    va_end(*ap*);
  }
  void *mp_print_str*(**MP** *mp*, **mp_string** *s*)
  {
    assert(*s* ≠ Λ);
    mp_do_print(*mp*, (**const char** *∗*) *s*→*str*, *s*→*len*);
  }

90.   Here is the very first thing that METAPOST prints: a headline that identifies the version number and base name. The *term_offset* variable is temporarily incorrect, but the discrepancy is not serious since we assume that the banner and mem identifier together will occupy at most *max_print_line* character positions.

⟨Initialize the output routines 81⟩ +≡
  wterm(*mp→banner*);
  mp_print_ln(*mp*);
  update_terminal( );

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

⟨Basic printing procedures 85⟩ +≡
  void *mp_print_nl*(**MP** *mp*, **const char** *∗s*)
  {   /∗ prints string *s* at beginning of line ∗/
    switch (*mp→selector*) {
    case *term_and_log*:
      if ((*mp→term_offset* > 0) ∨ (*mp→file_offset* > 0)) *mp_print_ln*(*mp*);
      break;
    case *log_only*:
      if (*mp→file_offset* > 0) *mp_print_ln*(*mp*);
      break;
    case *term_only*:
      if (*mp→term_offset* > 0) *mp_print_ln*(*mp*);
      break;
    case *no_print*: **case** *pseudo*: **case** *new_string*: **break**;
    }   /∗ there are no other cases ∗/
    mp_print(*mp*, *s*);
  }

92.   The following procedure, which prints out the decimal representation of a given integer *n*, assumes that all integers fit nicely into a **int**.

⟨Basic printing procedures 85⟩ +≡
  void *mp_print_int*(**MP** *mp*, **integer** *n*)
  {   /∗ prints an integer in decimal form ∗/
    char *s*[12];

    mp_snprintf(*s*, 12, "%d", (**int**) *n*);
    mp_print(*mp*, *s*);
  }
  void *mp_print_pointer*(**MP** *mp*, **void** *∗n*)
  {   /∗ prints an pointer in hexadecimal form ∗/
    char *s*[12];

    mp_snprintf(*s*, 12, "%p", *n*);
    mp_print(*mp*, *s*);
  }

93.   ⟨Internal library declarations 10⟩ +≡
  void *mp_print_int*(**MP** *mp*, **integer** *n*);
  void *mp_print_pointer*(**MP** *mp*, **void** *∗n*);

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

> **static void** mp_print_dd(**MP** mp, **integer** n)
> {   /∗ prints two least significant digits ∗/
> $n = abs(n) \% 100;$
> $mp\_print\_char(mp, xord(\text{'0'} + (n/10)));$
> $mp\_print\_char(mp, xord(\text{'0'} + (n \% 10)));$
> }

95.   ⟨Declarations 8⟩ +≡
> **static void** mp_print_dd(**MP** mp, **integer** n);

96.   Here is a procedure that asks the user to type a line of input, assuming that the *selector* setting is either *term_only* or *term_and_log*. The input is placed into locations *first* through *last* − 1 of the *buffer* array, and echoed on the transcript file if appropriate.

This procedure is never called when *interaction* < *mp_scroll_mode*.

> **#define** $prompt\_input(A)$ **do**
>   {
>   **if** (¬mp→$noninteractive$) {
> $wake\_up\_terminal();$
> $mp\_print(mp, (A));$
>   }
> $mp\_term\_input(mp);$
>   }
>   **while** (0)   /∗ prints a string and gets a line of input ∗/
>   **void** mp_term_input(**MP** mp)
>   {   /∗ gets a line from the terminal ∗/
>   **size_t** $k;$ /∗ index into *buffer* ∗/
>
>   **if** (mp→$noninteractive$) {
>   **if** (¬$mp\_input\_ln(mp, mp$→$term\_in)$) $longjmp(*(mp$→$jump\_buf), 1);$ /∗ chunk finished ∗/
> mp→$buffer[mp$→$last] = xord(\text{'%'});$
>   }
>   **else** {
> $update\_terminal();$ /∗ Now the user sees the prompt for sure ∗/
>   **if** (¬$mp\_input\_ln(mp, mp$→$term\_in)$) {
> $mp\_fatal\_error(mp, \texttt{"End\_of\_file\_on\_the\_terminal!"});$
>   }
> mp→$term\_offset = 0;$ /∗ the user's line ended with ⟨return⟩ ∗/
> $decr(mp$→$selector);$ /∗ prepare to echo the input ∗/
>   **if** (mp→$last \neq mp$→$first$) {
>   **for** ($k = mp$→$first;\ k < mp$→$last;\ k\text{++}$) {
> $mp\_print\_char(mp, mp$→$buffer[k]);$
>   }
>   }
> $mp\_print\_ln(mp);$
> mp→$buffer[mp$→$last] = xord(\text{'%'});$
> $incr(mp$→$selector);$ /∗ restore previous status ∗/
>   }
>   }

97. Reporting errors.

The *print_err* procedure supplies a '!' before the official message, and makes sure that the terminal is awake if a stop is going to occur. The **error** procedure supplies a '.' after the official message, then it shows the location of the error; and if *interaction* = *error_stop_mode*, it also enters into a dialog with the user, during which time the help message may be printed.

98. The global variable *interaction* has four settings, representing increasing amounts of user interaction:

⟨Exported types 15⟩ +≡
```
enum mp_interaction_mode {
    mp_unspecified_mode = 0,      /* extra value for command-line switch */
    mp_batch_mode,       /* omits all stops and omits terminal output */
    mp_nonstop_mode,       /* omits all stops */
    mp_scroll_mode,      /* omits error stops */
    mp_error_stop_mode       /* stops at every opportunity to interact */
};
```

99. ⟨Option variables 26⟩ +≡
```
int interaction;     /* current level of interaction */
int noninteractive;      /* do we have a terminal? */
```

100. Set it here so it can be overwritten by the commandline

⟨Allocate or initialize variables 28⟩ +≡
```
mp→interaction = opt→interaction;
if (mp→interaction ≡ mp_unspecified_mode ∨ mp→interaction > mp_error_stop_mode)
    mp→interaction = mp_error_stop_mode;
if (mp→interaction < mp_unspecified_mode) mp→interaction = mp_batch_mode;
```

101. *print_err* is not merged in **error** because it is also used in *prompt_file_name*, where **error** is not called at all.

⟨Declarations 8⟩ +≡
```
static void mp_print_err(MP mp, const char *A);
```

102.
```
static void mp_print_err(MP mp, const char *A){
    if (mp→interaction ≡ mp_error_stop_mode) wake_up_terminal();
    if (mp→file_line_error_style ∧ file_state ∧ ¬terminal_input) { mp_print_nl(mp, "");
    if (long_name ≠ Λ) {
        mp_print(mp, long_name);
    }
    else {
        mp_print(mp, mp_str(mp, name));
    }
    mp_print(mp, ":"); mp_print_int (mp, line ) ;
    mp_print(mp, ":␣"); }
    else {
        mp_print_nl(mp, "!␣");
    }
    mp_print(mp, A); }
```

103.   METAPOST is careful not to call **error** when the print *selector* setting might be unusual. The only possible values of *selector* at the time of error messages are

no_print (when *interaction* = *mp_batch_mode* and *log_file* not yet open);
term_only (when *interaction* > *mp_batch_mode* and *log_file* not yet open);
log_only (when *interaction* = *mp_batch_mode* and *log_file* is open);
term_and_log (when *interaction* > *mp_batch_mode* and *log_file* is open).

#define *initialize_print_selector*()
        $mp{\rightarrow}selector = (mp{\rightarrow}interaction \equiv mp\_batch\_mode$? *no_print* : *term_only*);

104.   The global variable *history* records the worst level of error that has been detected. It has four possible values: *spotless*, *warning_issued*, *error_message_issued*, and *fatal_error_stop*.

Another global variable, *error_count*, is increased by one when an **error** occurs without an interactive dialog, and it is reset to zero at the end of every statement. If *error_count* reaches 100, METAPOST decides that there is no point in continuing further.

⟨ Exported types 15 ⟩ +≡
  enum mp_history_state {
    mp_spotless = 0,   /∗ *history* value when nothing has been amiss yet ∗/
    mp_warning_issued,   /∗ *history* value when *begin_diagnostic* has been called ∗/
    mp_error_message_issued,   /∗ *history* value when **error** has been called ∗/
    mp_fatal_error_stop,   /∗ *history* value when termination was premature ∗/
    mp_system_error_stop   /∗ *history* value when termination was due to disaster ∗/
  };

105.   ⟨ Global variables 14 ⟩ +≡
  int *history*;   /∗ has the source input been clean so far? ∗/
  int *error_count*;   /∗ the number of scrolled errors since the last statement ended ∗/

106.   The value of *history* is initially *fatal_error_stop*, but it will be changed to *spotless* if METAPOST survives the initialization process.

107.   Since errors can be detected almost anywhere in METAPOST, we want to declare the error procedures near the beginning of the program. But the error procedures in turn use some other procedures, which need to be declared *forward* before we get to **error** itself.

It is possible for **error** to be called recursively if some error arises when *get_next* is being used to delete a token, and/or if some fatal error occurs while METAPOST is trying to fix a non-fatal one. But such recursion is never more than two levels deep.

⟨ Declarations 8 ⟩ +≡
  static void *mp_get_next*(**MP** *mp*);
  static void *mp_term_input*(**MP** *mp*);
  static void *mp_show_context*(**MP** *mp*);
  static void *mp_begin_file_reading*(**MP** *mp*);
  static void *mp_open_log_file*(**MP** *mp*);
  static void *mp_clear_for_error_prompt*(**MP** *mp*);

108.   ⟨ Internal library declarations 10 ⟩ +≡
  void *mp_normalize_selector*(**MP** *mp*);

109.   ⟨ Global variables 14 ⟩ +≡
  boolean *use_err_help*;   /∗ should the *err_help* string be shown? ∗/
  mp_string *err_help*;   /∗ a string set up by **errhelp** ∗/

110. ⟨Allocate or initialize variables 28⟩ +≡
  mp→*use_err_help* = *false*;

111. The *jump_out* procedure just cuts across all active procedure levels and goes to *end_of_MP*. This
is the only nonlocal **goto** statement in the whole program. It is used when there is no recovery from a
particular error.

  The program uses a *jump_buf* to handle this, this is initialized at three spots: the start of *mp_new*, the
start of *mp_initialize*, and the start of *mp_run*. Those are the only library enty points.

⟨Global variables 14⟩ +≡
  jmp_buf *∗jump_buf*;

112. If the array of internals is still Λ when *jump_out* is called, a crash occured during initialization, and
it is not safe to run the normal cleanup routine.

⟨Error handling procedures 112⟩ ≡
  void *mp_jump_out*(**MP** *mp*)
  {
    if (*mp*→*internal* ≠ Λ ∧ *mp*→*history* < *mp_system_error_stop*) *mp_close_files_and_terminate*(*mp*);
    longjmp(∗(*mp*→*jump_buf*), 1);
  }

See also sections 114, 132, 135, and 137.

This code is used in section 5.

113. ⟨Internal library declarations 10⟩ +≡
  void *mp_jump_out*(**MP** *mp*);

114.

⟨Error handling procedures 112⟩ +≡
  void *mp_warn*(**MP** *mp*, **const char** *∗msg*)
  {
    unsigned *saved_selector* = *mp*→*selector*;

    mp_normalize_selector(*mp*);
    mp_print_nl(*mp*, "Warning:␣");
    mp_print(*mp*, *msg*);
    mp_print_ln(*mp*);
    mp→*selector* = *saved_selector*;
  }

115.   Here now is the general **error** routine.

The argument *deletions_allowed* is set *false* if the *get_next* routine is active when **error** is called; this ensures that *get_next* will never be called recursively.

Individual lines of help are recorded in the array *help_line*, which contains entries in positions 0..(*help_ptr* − 1). They should be printed in reverse order, i.e., with *help_line*[0] appearing last.

```
void mp_error(MP mp, const char *msg, const char **hlp, boolean deletions_allowed)
{
   ASCII_code c;      /* what the user types */
   integer s1, s2;      /* used to save global variables when deleting tokens */
   mp_sym s3;      /* likewise */

   int i = 0;
   const char *help_line[6];      /* helps for the next error */
   unsigned int help_ptr;      /* the number of help lines present */
   const char **cnt = Λ;

   mp_print_err(mp, msg);
   if (hlp) {
      cnt = hlp;
      while (*cnt) {
         i++;
         cnt++;
      }
      cnt = hlp;
   }
   help_ptr = i;
   while (i > 0) {
      help_line[−−i] = *cnt++;
   }
   if (mp→history < mp_error_message_issued) mp→history = mp_error_message_issued;
   mp_print_char(mp, xord('.'));
   mp_show_context(mp);
   if (mp→halt_on_error) {
      mp→history = mp_fatal_error_stop;
      mp_jump_out(mp);
   }
   if ((¬mp→noninteractive) ∧ (mp→interaction ≡ mp_error_stop_mode)) {
      ⟨ Get user's advice and return 117 ⟩;
   }
   incr(mp→error_count);
   if (mp→error_count ≡ 100) {
      mp_print_nl(mp, "(That␣makes␣100␣errors;␣please␣try␣again.)");
      ;
      mp→history = mp_fatal_error_stop;
      mp_jump_out(mp);
   }
   ⟨ Put help message on the transcript file 130 ⟩;
}
```

116.   ⟨ Exported function headers 18 ⟩ +≡
```
extern void mp_error(MP mp, const char *msg, const char **hlp, boolean deletions_allowed);
extern void mp_warn(MP mp, const char *msg);
```

117.   ⟨Get user's advice and **return** 117⟩ ≡
  while (*true*) {
  CONTINUE: *mp_clear_for_error_prompt*(*mp*);
    prompt_input("?␣");
    ;
    if (*mp*→*last* ≡ *mp*→*first*) **return**;
    c = *mp*→*buffer*[*mp*→*first*];
    if (*c* ≥ 'a') *c* = (**ASCII_code**)(*c* + 'A' − 'a');   /∗ convert to uppercase ∗/
    ⟨Interpret code *c* and **return** if done 123⟩;
  }

This code is used in section 115.

118.   It is desirable to provide an 'E' option here that gives the user an easy way to return from META-
POST to the system editor, with the offending line ready to be edited. But such an extension requires some
system wizardry, so the present implementation simply types out the name of the file that should be edited
and the relevant line number.
⟨Exported types 15⟩ +≡
  typedef void(∗*mp_editor_cmd*)(**MP**, **char** ∗, **int**);

119.   ⟨Option variables 26⟩ +≡
  mp_editor_cmd run_editor;

120.   ⟨Allocate or initialize variables 28⟩ +≡
  set_callback_option(*run_editor*);

121.   ⟨Declarations 8⟩ +≡
  static void *mp_run_editor*(**MP** *mp*, **char** ∗*fname*, **int** *fline*);

122.   **void** *mp_run_editor*(**MP** *mp*, **char** ∗*fname*, **int** *fline*)
  {
    char ∗*s* = *xmalloc*(256, 1);

    mp_snprintf(*s*, 256, "You␣want␣to␣edit␣file␣%s␣at␣line␣%d\n", *fname*, *fline*);
    wterm_ln(*s*);
  }

123.

⟨Interpret code c and **return** if done 123⟩ ≡

  switch (c) {

  case '0': **case** '1': **case** '2': **case** '3': **case** '4': **case** '5': **case** '6': **case** '7': **case** '8':

    case '9':

    if (*deletions_allowed*) {

      ⟨Delete tokens and **continue** 127⟩;

    }

    break;

  case 'E':

    if $(mp\text{→}file\_ptr > 0)$ {

      $mp\text{→}interaction = mp\_scroll\_mode$;

      $mp\_close\_files\_and\_terminate(mp)$;

      $(mp\text{→}run\_editor)(mp, mp\_str(mp, mp\text{→}input\_stack[mp\text{→}file\_ptr].name\_field), mp\_true\_line(mp))$;

      $mp\_jump\_out(mp)$;

    }

    break;

  case 'H': ⟨Print the help information and **continue** 128⟩;   /∗ **break**; ∗/

  case 'I': ⟨Introduce new material from the terminal and **return** 126⟩;   /∗ **break**; ∗/

  case 'Q': **case** 'R': **case** 'S': ⟨Change the interaction level and **return** 125⟩;   /∗ **break**; ∗/

  case 'X': $mp\text{→}interaction = mp\_scroll\_mode$;

    $mp\_jump\_out(mp)$;

    break;

  default: **break**;

  }

  ⟨Print the menu of available options 124⟩

This code is used in section 117.

124. ⟨Print the menu of available options 124⟩ ≡

  {

    $mp\_print(mp, \text{"Type}_\sqcup\text{<return>}_\sqcup\text{to}_\sqcup\text{proceed,}_\sqcup\text{S}_\sqcup\text{to}_\sqcup\text{scroll}_\sqcup\text{future}_\sqcup\text{error}_\sqcup\text{messages,"})$;

    ;

    $mp\_print\_nl(mp, \text{"R}_\sqcup\text{to}_\sqcup\text{run}_\sqcup\text{without}_\sqcup\text{stopping,}_\sqcup\text{Q}_\sqcup\text{to}_\sqcup\text{run}_\sqcup\text{quietly,"})$;

    $mp\_print\_nl(mp, \text{"I}_\sqcup\text{to}_\sqcup\text{insert}_\sqcup\text{something,}_\sqcup\text{"})$;

    if $(mp\text{→}file\_ptr > 0)$ $mp\_print(mp, \text{"E}_\sqcup\text{to}_\sqcup\text{edit}_\sqcup\text{your}_\sqcup\text{file,"})$;

    if (*deletions_allowed*)

      $mp\_print\_nl(mp, \text{"1}_\sqcup\text{or}_\sqcup\text{...}_\sqcup\text{or}_\sqcup\text{9}_\sqcup\text{to}_\sqcup\text{ignore}_\sqcup\text{the}_\sqcup\text{next}_\sqcup\text{1}_\sqcup\text{to}_\sqcup\text{9}_\sqcup\text{tokens}_\sqcup\text{of}_\sqcup\text{input,"})$;

    $mp\_print\_nl(mp, \text{"H}_\sqcup\text{for}_\sqcup\text{help,}_\sqcup\text{X}_\sqcup\text{to}_\sqcup\text{quit."})$;

  }

This code is used in section 123.

125.   ⟨Change the interaction level and **return** 125⟩ ≡
```
  {
    mp→error_count = 0;
    mp_print(mp, "OK,␣entering␣");
    switch (c) {
    case 'Q': mp→interaction = mp_batch_mode;
      mp_print(mp, "batchmode");
      decr(mp→selector);
      break;
    case 'R': mp→interaction = mp_nonstop_mode;
      mp_print(mp, "nonstopmode");
      break;
    case 'S': mp→interaction = mp_scroll_mode;
      mp_print(mp, "scrollmode");
      break;
    }    /* there are no other cases */
    mp_print(mp, "...");
    mp_print_ln(mp);
    update_terminal();
    return;
  }
```
This code is used in section 123.

126.   When the following code is executed, $buffer[(first+1) .. (last-1)]$ may contain the material inserted by the user; otherwise another prompt will be given. In order to understand this part of the program fully, you need to be familiar with METAPOST's input stacks.

⟨Introduce new material from the terminal and **return** 126⟩ ≡
```
  {
    mp_begin_file_reading(mp);       /* enter a new syntactic level for terminal input */
    if (mp→last > mp→first + 1) {
      loc = (halfword)(mp→first + 1);
      mp→buffer[mp→first] = xord('␣');
    }
    else {
      prompt_input("insert>");
      loc = (halfword)mp→first;
    }
    mp→first = mp→last + 1;
    mp→cur_input.limit_field = (halfword)mp→last;
    return;
  }
```
This code is used in section 123.

127.   We allow deletion of up to 99 tokens at a time.

⟨ Delete tokens and **continue** 127 ⟩ ≡
  {
    $s1 = cur\_cmd(\,)$;
    $s2 = cur\_mod(\,)$;
    $s3 = cur\_sym(\,)$;
    $mp{\rightarrow}OK\_to\_interrupt = false$;
    if $((mp{\rightarrow}last > mp{\rightarrow}first + 1) \wedge (mp{\rightarrow}buffer\,[mp{\rightarrow}first + 1] \geq \text{'0'}) \wedge (mp{\rightarrow}buffer\,[mp{\rightarrow}first + 1] \leq \text{'9'}))$
      $c = xord\,(c * 10 + mp{\rightarrow}buffer\,[mp{\rightarrow}first + 1] - \text{'0'} * 11)$;
    else $c = (\mathbf{ASCII\_code})(c - \text{'0'})$;
    while $(c > 0)$ {
      $mp\_get\_next\,(mp)$;   /* one-level recursive call of **error** is possible */
      ⟨ Decrease the string reference count, if the current token is a string 812 ⟩;
      $c{-}{-}$;
    }
    ;
    $set\_cur\_cmd\,(s1)$;
    $set\_cur\_mod\,(s2)$;
    $set\_cur\_sym\,(s3)$;
    $mp{\rightarrow}OK\_to\_interrupt = true$;
    $help\_ptr = 2$;
    $help\_line\,[1] = \text{"I}_\sqcup\text{have}_\sqcup\text{just}_\sqcup\text{deleted}_\sqcup\text{some}_\sqcup\text{text},_\sqcup\text{as}_\sqcup\text{you}_\sqcup\text{asked."}$;
    $help\_line\,[0] = \text{"You}_\sqcup\text{can}_\sqcup\text{now}_\sqcup\text{delete}_\sqcup\text{more},_\sqcup\text{or}_\sqcup\text{insert},_\sqcup\text{or}_\sqcup\text{whatever."}$;
    $mp\_show\_context\,(mp)$;
    goto CONTINUE;
  }

This code is used in section 123.

128.   Some wriggling with *help_line* is done here to avoid giving no information whatsoever, or presenting the same information twice in a row.

⟨ Print the help information and **continue** 128 ⟩ ≡
```
{
  if (mp→use_err_help) {
    ⟨ Print the string err_help, possibly on several lines 129 ⟩;
    mp→use_err_help = false;
  }
  else {
    if (help_ptr ≡ 0) {
      help_ptr = 2;
      help_line[1] = "Sorry,␣I␣don't␣know␣how␣to␣help␣in␣this␣situation.";
      help_line[0] = "Maybe␣you␣should␣try␣asking␣a␣human?";
    }
    do {
      decr(help_ptr);
      mp_print(mp, help_line[help_ptr]);
      mp_print_ln(mp);
    } while (help_ptr ≠ 0);
  }
  ;
  help_ptr = 4;
  help_line[3] = "Sorry,␣I␣already␣gave␣what␣help␣I␣could...";
  help_line[2] = "Maybe␣you␣should␣try␣asking␣a␣human?";
  help_line[1] = "An␣error␣might␣have␣occurred␣before␣I␣noticed␣any␣problems.";
  help_line[0] = "``If␣all␣else␣fails,␣read␣the␣instructions.''";
  goto CONTINUE;
}
```
This code is used in section 123.

129.   ⟨ Print the string *err_help*, possibly on several lines 129 ⟩ ≡
```
{
  size_t j = 0;
  while (j < mp→err_help→len) {
    if (*(mp→err_help→str + j) ≠ '%') mp_print(mp, (const char *)(mp→err_help→str + j));
    else if (j + 1 ≡ mp→err_help→len) mp_print_ln(mp);
    else if (*(mp→err_help→str + j) ≠ '%') mp_print_ln(mp);
    else {
      j++;
      mp_print_char(mp, xord('%'));
    }
    ;
    j++;
  }
}
```
This code is used in sections 128 and 130.

130.   ⟨Put help message on the transcript file 130⟩ ≡
  if (*mp⃗interaction* > *mp_batch_mode*) *decr*(*mp⃗selector*);   /∗ avoid terminal output ∗/
  if (*mp⃗use_err_help*) {
    mp_print_nl(*mp*, "");
    ⟨Print the string *err_help*, possibly on several lines 129⟩;
  }
  else {
    while (*help_ptr* > 0) {
      decr(*help_ptr*);
      mp_print_nl(*mp*, *help_line*[*help_ptr*]);
    }
    ;
    mp_print_ln(*mp*);
    if (*mp⃗interaction* > *mp_batch_mode*) *incr*(*mp⃗selector*);   /∗ re-enable terminal output ∗/
    mp_print_ln(*mp*);
  }

This code is used in section 115.

131.   In anomalous cases, the print selector might be in an unknown state; the following subroutine is
called to fix things just enough to keep running a bit longer.
  void *mp_normalize_selector*(**MP** *mp*)
  {
    if (*mp⃗log_opened*) *mp⃗selector* = *term_and_log*;
    else *mp⃗selector* = *term_only*;
    if (*mp⃗job_name* ≡ Λ) *mp_open_log_file*(*mp*);
    if (*mp⃗interaction* ≡ *mp_batch_mode*) *decr*(*mp⃗selector*);
  }

132.   The following procedure prints METAPOST's last words before dying.
⟨Error handling procedures 112⟩ +≡
  void *mp_fatal_error*(**MP** *mp*, **const char** ∗*s*)
  {   /∗ prints *s*, and that's it ∗/
    const char ∗*hlp*[ ] = {*s*, Λ};
    mp_normalize_selector(*mp*);
    if (*mp⃗interaction* ≡ *mp_error_stop_mode*) *mp⃗interaction* = *mp_scroll_mode*;
        /∗ no more interaction ∗/
    if (*mp⃗log_opened*) *mp_error*(*mp*, "Emergency␣stop", *hlp*, *true*);
    mp⃗history = *mp_fatal_error_stop*;
    mp_jump_out(*mp*);   /∗ irrecoverable error ∗/
  }

133.   ⟨Exported function headers 18⟩ +≡
  extern void *mp_fatal_error*(**MP** *mp*, **const char** ∗*s*);

134.   ⟨Internal library declarations 10⟩ +≡
  void *mp_overflow*(**MP** *mp*, **const char** ∗*s*, **integer** *n*);

135.   ⟨Error handling procedures 112⟩ +≡
  void *mp_overflow*(**MP** *mp*, **const char** *∗s*, **integer** *n*)
  {   /∗ stop due to finiteness ∗/
    char *msg*[256];
    const char *∗hlp*[ ] = {"If␣you␣really␣absolutely␣need␣more␣capacity,",
        "you␣can␣ask␣a␣wizard␣to␣enlarge␣me.", Λ};
    mp_normalize_selector(*mp*);
    mp_snprintf(*msg*, 256, "MetaPost␣capacity␣exceeded,␣sorry␣[%s=%d]", *s*, (**int**) *n*);
    ;
    if (*mp*→*interaction* ≡ *mp_error_stop_mode*) *mp*→*interaction* = *mp_scroll_mode*;
        /∗ no more interaction ∗/
    if (*mp*→*log_opened*) *mp_error*(*mp*, *msg*, *hlp*, *true*);
    mp→*history* = *mp_fatal_error_stop*;
    mp_jump_out(*mp*);   /∗ irrecoverable error ∗/
  }

136.   The program might sometime run completely amok, at which point there is no choice but to stop. If no previous error has been detected, that's bad news; a message is printed that is really intended for the METAPOST maintenance person instead of the user (unless the user has been particularly diabolical). The index entries for 'this can't happen' may help to pinpoint the problem.

⟨Internal library declarations 10⟩ +≡
  void *mp_confusion*(**MP** *mp*, **const char** *∗s*);

137.   Consistency check violated; *s* tells where.

⟨Error handling procedures 112⟩ +≡
  void *mp_confusion*(**MP** *mp*, **const char** *∗s*)
  {
    char *msg*[256];
    const char *∗hlp*[ ] = {"One␣of␣your␣faux␣pas␣seems␣to␣have␣wounded␣me␣deeply...",
        "in␣fact,␣I'm␣barely␣conscious.␣Please␣fix␣it␣and␣try␣again.", Λ};
    mp_normalize_selector(*mp*);
    if (*mp*→*history* < *mp_error_message_issued*) {
      mp_snprintf(*msg*, 256, "This␣can't␣happen␣(%s)", *s*);
      ;
      hlp[0] = "I'm␣broken.␣Please␣show␣this␣to␣someone␣who␣can␣fix␣can␣fix";
      hlp[1] = Λ;
    }
    else {
      mp_snprintf(*msg*, 256, "I␣can\'t␣go␣on␣meeting␣you␣like␣this");
      ;
    }
    if (*mp*→*interaction* ≡ *mp_error_stop_mode*) *mp*→*interaction* = *mp_scroll_mode*;
        /∗ no more interaction ∗/
    if (*mp*→*log_opened*) *mp_error*(*mp*, *msg*, *hlp*, *true*);
    mp→*history* = *mp_fatal_error_stop*;
    mp_jump_out(*mp*);   /∗ irrecoverable error ∗/
  }

138.   Users occasionally want to interrupt METAPOST while it's running. If the runtime system allows
this, one can implement a routine that sets the global variable *interrupt* to some nonzero value when such
an interrupt is signaled. Otherwise there is probably at least a way to make *interrupt* nonzero using the C
debugger.

#define   *check_interrupt*
          {
              if (*mp→interrupt* ≠ 0) *mp_pause_for_instructions*(*mp*);
          }
⟨Global variables 14⟩ +≡
   integer *interrupt*;   /∗ should METAPOST pause for instructions? ∗/
   boolean *OK_to_interrupt*;   /∗ should interrupts be observed? ∗/
   integer *run_state*;   /∗ are we processing input ? ∗/
   boolean *finished*;   /∗ set true by *close_files_and_terminate* ∗/
   boolean *reading_preload*;

139.   ⟨Allocate or initialize variables 28⟩ +≡
   mp→OK_to_interrupt = *true*;
   mp→finished = *false*;

140.   When an interrupt has been detected, the program goes into its highest interaction level and lets
the user have the full flexibility of the **error** routine. METAPOST checks for interrupts only at times when
it is safe to do this.

   static void *mp_pause_for_instructions*(**MP** *mp*)
   {
      const char ∗*hlp*[] = {"You␣rang?",
          "Try␣to␣insert␣some␣instructions␣for␣me␣(e.g.,'I␣show␣x'),",
          "unless␣you␣just␣want␣to␣quit␣by␣typing␣'X'.", Λ};
      if (*mp→OK_to_interrupt*) {
         mp→interaction = *mp_error_stop_mode*;
         if ((*mp→selector* ≡ *log_only*) ∨ (*mp→selector* ≡ *no_print*)) *incr*(*mp→selector*);
         ;
         mp_error(*mp*, "Interruption", *hlp*, *false*);
         mp→interrupt = 0;
      }
   }

141.   Arithmetic with scaled numbers.   The principal computations performed by METAPOST are done entirely in terms of integers less than 2^{31} in magnitude; thus, the arithmetic specified in this program can be carried out in exactly the same way on a wide variety of computers, including some small ones.

But C does not rigidly define the / operation in the case of negative dividends; for example, the result of $(-2*n-1)/2$ is $-(n+1)$ on some computers and $-n$ on others (is this true ?). There are two principal types of arithmetic: "translation-preserving," in which the identity $(a+q*b)/b = (a/b)+q$ is valid; and "negation-preserving," in which $(-a)/b = -(a/b)$. This leads to two METAPOSTs, which can produce different results, although the differences should be negligible when the language is being used properly. The TEX processor has been defined carefully so that both varieties of arithmetic will produce identical output, but it would be too inefficient to constrain METAPOST in a similar way.

#define *inf_t* $((\textbf{math\_data}\ *)\ mp{\to}math){\to}inf\_t$

142.   A single computation might use several subroutine calls, and it is desirable to avoid producing multiple error messages in case of arithmetic overflow. So the routines below set the global variable *arith_error* to *true* instead of reporting errors directly to the user.

⟨ Global variables 14 ⟩ +≡
  boolean *arith_error*;   /* has arithmetic overflow occurred recently? */

143.   ⟨ Allocate or initialize variables 28 ⟩ +≡
  $mp{\to}arith\_error = false;$

144.   At crucial points the program will say *check_arith*, to test if an arithmetic error has been detected.

#define *check_arith*() **do**
        {
          if (*mp{\to}arith_error*) *mp_clear_arith*(*mp*);
        }
        while (0)
  static void *mp_clear_arith*(**MP** *mp*)
  {
    const char *$*hlp[\,] = \{$"Uh,␣oh.␣A␣little␣while␣ago␣one␣of␣the␣quantities␣that␣I␣was",
        "computing␣got␣too␣large,␣so␣I'm␣afraid␣your␣answers␣will␣be",
        "somewhat␣askew.␣You'll␣probably␣have␣to␣adopt␣different",
        "tactics␣next␣time.␣But␣I␣shall␣try␣to␣carry␣on␣anyway."$, \Lambda\};$
    $mp_error(mp, $"Arithmetic␣overflow"$, hlp, true);$
    ;
    $mp{\to}arith\_error = false;$
  }

145.   The definitions of these are set up by the math initialization.

#define *arc_tol_k*   ((**math_data** ∗) *mp*→*math*)→*arc_tol_k*
#define *coef_bound_k*   ((**math_data** ∗) *mp*→*math*)→*coef_bound_k*
#define *coef_bound_minus_1*   ((**math_data** ∗) *mp*→*math*)→*coef_bound_minus_1*
#define *sqrt_8_e_k*   ((**math_data** ∗) *mp*→*math*)→*sqrt_8_e_k*
#define *twelve_ln_2_k*   ((**math_data** ∗) *mp*→*math*)→*twelve_ln_2_k*
#define *twelvebits_3*   ((**math_data** ∗) *mp*→*math*)→*twelvebits_3*
#define *one_k*   ((**math_data** ∗) *mp*→*math*)→*one_k*
#define *epsilon_t*   ((**math_data** ∗) *mp*→*math*)→*epsilon_t*
#define *unity_t*   ((**math_data** ∗) *mp*→*math*)→*unity_t*
#define *zero_t*   ((**math_data** ∗) *mp*→*math*)→*zero_t*
#define *two_t*   ((**math_data** ∗) *mp*→*math*)→*two_t*
#define *three_t*   ((**math_data** ∗) *mp*→*math*)→*three_t*
#define *half_unit_t*   ((**math_data** ∗) *mp*→*math*)→*half_unit_t*
#define *three_quarter_unit_t*   ((**math_data** ∗) *mp*→*math*)→*three_quarter_unit_t*
#define *twentysixbits_sqrt2_t*   ((**math_data** ∗) *mp*→*math*)→*twentysixbits_sqrt2_t*
#define *twentyeightbits_d_t*   ((**math_data** ∗) *mp*→*math*)→*twentyeightbits_d_t*
#define *twentysevenbits_sqrt2_d_t*   ((**math_data** ∗) *mp*→*math*)→*twentysevenbits_sqrt2_d_t*
#define *warning_limit_t*   ((**math_data** ∗) *mp*→*math*)→*warning_limit_t*
#define *precision_default*   ((**math_data** ∗) *mp*→*math*)→*precision_default*
#define *precision_max*   ((**math_data** ∗) *mp*→*math*)→*precision_max*
#define *precision_min*   ((**math_data** ∗) *mp*→*math*)→*precision_min*

146.   In fact, the two sorts of scaling discussed above aren't quite sufficient; METAPOST has yet another, used internally to keep track of angles.

147.   We often want to print two scaled quantities in parentheses, separated by a comma.
⟨Basic printing procedures 85⟩ +≡
  void *mp_print_two*(**MP** *mp*, **mp_number** *x*, **mp_number** *y*)
  {   /∗ prints '(*x*, *y*)' ∗/
    mp_print_char(*mp*, *xord*('('));
    print_number(*x*);
    mp_print_char(*mp*, *xord*(','));
    print_number(*y*);
    mp_print_char(*mp*, *xord*(')'));
  }

148.

#define *fraction_one_t*   ((**math_data** ∗) *mp*→*math*)→*fraction_one_t*
#define *fraction_half_t*   ((**math_data** ∗) *mp*→*math*)→*fraction_half_t*
#define *fraction_three_t*   ((**math_data** ∗) *mp*→*math*)→*fraction_three_t*
#define *fraction_four_t*   ((**math_data** ∗) *mp*→*math*)→*fraction_four_t*
#define *one_eighty_deg_t*   ((**math_data** ∗) *mp*→*math*)→*one_eighty_deg_t*
#define *three_sixty_deg_t*   ((**math_data** ∗) *mp*→*math*)→*three_sixty_deg_t*

149.   ⟨Local variables for initialization 35⟩ +≡
  integer *k*;   /∗ all-purpose loop index ∗/

150.   And now let's complete our collection of numeric utility routines by considering random number generation.   METAPOST generates pseudo-random numbers with the additive scheme recommended in Section 3.6 of *The Art of Computer Programming*; however, the results are random fractions between 0 and *fraction_one* − 1, inclusive.

There's an auxiliary array *randoms* that contains 55 pseudo-random fractions.   Using the recurrence $x_n = (x_{n-55} - x_{n-31}) \bmod 2^{28}$, we generate batches of 55 new x_n's at a time by calling *new_randoms*. The global variable *j_random* tells which element has most recently been consumed. The global variable *random_seed* was introduced in version 0.9, for the sole reason of stressing the fact that the initial value of the random seed is system-dependant.   The initialization code below will initialize this variable to (*internal*[*mp_time*] *div unity*) + *internal*[*mp_day*], but this is not good enough on modern fast machines that are capable of running multiple MetaPost processes within the same second.

⟨ Global variables 14 ⟩ +≡
  mp_number *randoms*[55];   /∗ the last 55 random values generated ∗/
  int *j_random*;   /∗ the number of unused *randoms* ∗/

151.   ⟨ Option variables 26 ⟩ +≡
  int *random_seed*;   /∗ the default random seed ∗/

152.   ⟨ Allocate or initialize variables 28 ⟩ +≡
  mp→*random_seed* = *opt*→*random_seed*;
  {
    int *i*;
    for (*i* = 0; *i* < 55; *i*++) {
      new_fraction(*mp*→*randoms*[*i*]);
    }
  }

153.   ⟨ Dealloc variables 27 ⟩ +≡
  {
    int *i*;
    for (*i* = 0; *i* < 55; *i*++) {
      free_number(*mp*→*randoms*[*i*]);
    }
  }

154.   ⟨ Internal library declarations 10 ⟩ +≡
  void *mp_new_randoms*(**MP** *mp*);

155.   **void** $mp\_new\_randoms(\mathbf{MP}\ mp)$
{
   int k;   /∗ index into $randoms$ ∗/
   mp_number x;   /∗ accumulator ∗/
   $new\_number(x)$;
   for $(k = 0;\ k \leq 23;\ k{+}{+})$ {
     $set\_number\_from\_substraction(x, mp{\rightarrow}randoms[k], mp{\rightarrow}randoms[k + 31])$;
     if $(number\_negative(x))$ $number\_add(x, fraction\_one\_t)$;
     $number\_clone(mp{\rightarrow}randoms[k], x)$;
   }
   for $(k = 24;\ k \leq 54;\ k{+}{+})$ {
     $set\_number\_from\_substraction(x, mp{\rightarrow}randoms[k], mp{\rightarrow}randoms[k - 24])$;
     if $(number\_negative(x))$ $number\_add(x, fraction\_one\_t)$;
     $number\_clone(mp{\rightarrow}randoms[k], x)$;
   }
   $free\_number(x)$;
   $mp{\rightarrow}j\_random = 54$;
}

156.   To consume a random fraction, the program below will say '$next\_random$'.

static void $mp\_next\_random(\mathbf{MP}\ mp, \mathbf{mp\_number}\ {*}ret)$
{
   if $(mp{\rightarrow}j\_random \equiv 0)$ $mp\_new\_randoms(mp)$;
   else $decr(mp{\rightarrow}j\_random)$;
   $number\_clone({*}ret, mp{\rightarrow}randoms[mp{\rightarrow}j\_random])$;
}

157.   To produce a uniform random number in the range $0 \leq u < x$ or $0 \geq u > x$ or $0 = u = x$, given a *scaled* value x, we proceed as shown here.

Note that the call of *take_fraction* will produce the values 0 and x with about half the probability that it will produce any other particular values between 0 and x, because it rounds its answers.

```
static void mp_unif_rand(MP mp, mp_number *ret, mp_number x_orig)
{
  mp_number y;     /* trial value */
  mp_number x, abs_x;
  mp_number u;
  new_fraction(y);
  new_number(x);
  new_number(abs_x);
  new_number(u);
  number_clone(x, x_orig);
  number_clone(abs_x, x);
  number_abs(abs_x);
  mp_next_random(mp, &u);
  take_fraction(y, abs_x, u);
  free_number(u);
  if (number_equal(y, abs_x)) {
    set_number_to_zero(*ret);
  }
  else if (number_positive(x)) {
    number_clone(*ret, y);
  }
  else {
    number_clone(*ret, y);
    number_negate(*ret);
  }
  free_number(abs_x);
  free_number(x);
  free_number(y);
}
```

158.   Finally, a normal deviate with mean zero and unit standard deviation can readily be obtained with the ratio method (Algorithm 3.4.1R in *The Art of Computer Programming*).

```
static void mp_norm_rand(MP mp, mp_number *ret)
{
  mp_number ab_vs_cd;
  mp_number abs_x;
  mp_number u;
  mp_number r;
  mp_number la, xa;

  new_number(ab_vs_cd);
  new_number(la);
  new_number(xa);
  new_number(abs_x);
  new_number(u);
  new_number(r);
  do {
    do {
      mp_number v;

      new_number(v);
      mp_next_random(mp, &v);
      number_substract(v, fraction_half_t);
      take_fraction(xa, sqrt_8_e_k, v);
      free_number(v);
      mp_next_random(mp, &u);
      number_clone(abs_x, xa);
      number_abs(abs_x);
    } while (number_greaterequal(abs_x, u));
    make_fraction(r, xa, u);
    number_clone(xa, r);
    m_log(la, u);
    set_number_from_substraction(la, twelve_ln_2_k, la);
    ab_vs_cd(ab_vs_cd, one_k, la, xa, xa);
  } while (number_negative(ab_vs_cd));
  number_clone(*ret, xa);
  free_number(ab_vs_cd);
  free_number(r);
  free_number(abs_x);
  free_number(la);
  free_number(xa);
  free_number(u);
}
```

159.   Packed data.

#define $max\_quarterword$ #**3FFF**   /∗ largest allowable value in a *quarterword* ∗/
#define $max\_halfword$ #**FFFFFFF**   /∗ largest allowable value in a *halfword* ∗/

160.   The macros *qi* and *qo* are used for input to and output from quarterwords. These are legacy macros.

#define $qo(A)$ (A) /∗ to read eight bits from a quarterword ∗/
#define $qi(A)$ (*quarterword*)(A) /∗ to store eight bits in a quarterword ∗/

161.   The reader should study the following definitions closely:

⟨ Types in the outer block 33 ⟩ +≡
  typedef struct $mp\_value\_node\_data$ ∗**mp_value_node**;
  typedef struct mp_node_data ∗**mp_node**;
  typedef struct mp_symbol_entry ∗**mp_sym**;
  typedef short quarterword;   /∗ 1/4 of a word ∗/
  typedef int halfword;   /∗ 1/2 of a word ∗/
  typedef struct {
    integer *scale*;   /∗ only for *indep_scale*, used together with *serial* ∗/
    integer *serial*;   /∗ only for *indep_value*, used together with *scale* ∗/
  } **mp_independent_data**;
  typedef struct {
    mp_independent_data *indep*;
    mp_number *n*;
    mp_string *str*;
    mp_sym *sym*;
    mp_node *node*;

    $mp\_knot\,p$;
  } **mp_value_data**;
  typedef struct {
    $mp\_variable\_type\,type$;

    mp_value_data *data*;
  } **mp_value**;
  typedef struct {
    quarterword *b0*, *b1*, *b2*, *b3*;
  } **four_quarters**;
  typedef union {
    integer *sc*;
    four_quarters *qqqq*;
  } **font_data**;

162.   The global variable *math_mode* has four settings, representing the math value type that will be used in this run.
  the typedef for **mp_number** is here because it has to come very early.

⟨ Exported types 15 ⟩ +≡
  typedef enum {
    $mp\_math\_scaled\_mode = 0, mp\_math\_double\_mode = 1, mp\_math\_binary\_mode = 2,$
      $mp\_math\_decimal\_mode = 3$
  } **mp_math_mode**;

163.   ⟨ Option variables 26 ⟩ +≡
  int *math_mode*;   /∗ math mode ∗/

164.   ⟨Allocate or initialize variables 28⟩ +≡
  $mp\text{-}math\_mode = opt\text{-}math\_mode$;

165.
#define $xfree(A)$ **do**
        {
          $mp\_xfree(A)$;
          $A = \Lambda$;
        }
        while (0)
#define $xrealloc(P, A, B)$ $mp\_xrealloc(mp, P, (\textbf{size\_t})\ A, B)$
#define $xmalloc(A, B)$ $mp\_xmalloc(mp, (\textbf{size\_t})\ A, B)$
#define $xstrdup(A)$ $mp\_xstrdup(mp, A)$
#define $\texttt{XREALLOC}(a, b, c)$ $a = xrealloc(a, (b + 1), \textbf{sizeof}\ (c))$;

⟨Declare helpers 165⟩ ≡
  extern void $mp\_xfree(\textbf{void}\ *x)$;
  extern void $*mp\_xrealloc(\textbf{MP}\ mp, \textbf{void}\ *p, \textbf{size\_t}\ nmem, \textbf{size\_t}\ size)$;
  extern void $*mp\_xmalloc(\textbf{MP}\ mp, \textbf{size\_t}\ nmem, \textbf{size\_t}\ size)$;
  extern void $mp\_do\_snprintf(\textbf{char}\ *str, \textbf{int}\ size, \textbf{const char}\ *fmt, \ldots)$;
  extern void $*do\_alloc\_node(\textbf{MP}\ mp, \textbf{size\_t}\ size)$;
This code is used in section 4.

166.   This is an attempt to spend less time in $malloc(\ )$:
#define $max\_num\_token\_nodes$ 1000
#define $max\_num\_pair\_nodes$ 1000
#define $max\_num\_knot\_nodes$ 1000
#define $max\_num\_value\_nodes$ 1000
#define $max\_num\_symbolic\_nodes$ 1000
⟨Global variables 14⟩ +≡
  mp_node $token\_nodes$;
  int num_token_nodes;
  mp_node $pair\_nodes$;
  int num_pair_nodes;

  $mp\_knot\ knot\_nodes$;

  int num_knot_nodes;
  mp_node $value\_nodes$;
  int num_value_nodes;
  mp_node $symbolic\_nodes$;
  int $num\_symbolic\_nodes$;

167. ⟨Allocate or initialize variables 28⟩ +≡
$mp \rightarrow token\_nodes = \Lambda$;
$mp \rightarrow num\_token\_nodes = 0$;
$mp \rightarrow pair\_nodes = \Lambda$;
$mp \rightarrow num\_pair\_nodes = 0$;
$mp \rightarrow knot\_nodes = \Lambda$;
$mp \rightarrow num\_knot\_nodes = 0$;
$mp \rightarrow value\_nodes = \Lambda$;
$mp \rightarrow num\_value\_nodes = 0$;
$mp \rightarrow symbolic\_nodes = \Lambda$;
$mp \rightarrow num\_symbolic\_nodes = 0$;

168. ⟨Dealloc variables 27⟩ +≡
while ($mp \rightarrow value\_nodes$) {
    mp_node $p = mp \rightarrow value\_nodes$;
    $mp \rightarrow value\_nodes = p \rightarrow link$;
    $mp\_free\_node(mp, p, value\_node\_size)$;
}
while ($mp \rightarrow symbolic\_nodes$) {
    mp_node $p = mp \rightarrow symbolic\_nodes$;
    $mp \rightarrow symbolic\_nodes = p \rightarrow link$;
    $mp\_free\_node(mp, p, symbolic\_node\_size)$;
}
while ($mp \rightarrow pair\_nodes$) {
    mp_node $p = mp \rightarrow pair\_nodes$;
    $mp \rightarrow pair\_nodes = p \rightarrow link$;
    $mp\_free\_node(mp, p, pair\_node\_size)$;
}
while ($mp \rightarrow token\_nodes$) {
    mp_node $p = mp \rightarrow token\_nodes$;
    $mp \rightarrow token\_nodes = p \rightarrow link$;
    $mp\_free\_node(mp, p, token\_node\_size)$;
}
while ($mp \rightarrow knot\_nodes$) {
    $mp\_knot\, p = mp \rightarrow knot\_nodes$;
    $mp \rightarrow knot\_nodes = p \rightarrow next$;
    $mp\_free\_knot(mp, p)$;
}

169.   This is a nicer way of allocating nodes.
#define $malloc\_node(A)$ $do\_alloc\_node(mp, (A))$

170.

```
void *do_alloc_node(MP mp, size_t size)
{
  void *p;
  p = xmalloc(1, size);
  add_var_used(size);
  ((mp_node) p)→link = Λ;
  ((mp_node) p)→has_number = 0;
  return p;
}
```

171.   The *max_size_test* guards against overflow, on the assumption that **size_t** is at least 31bits wide.

#**define** *max_size_test* #7FFFFFFF

```
  void mp_xfree(void *x)
  {
    if (x ≠ Λ) free(x);
  }
  void *mp_xrealloc(MP mp, void *p, size_t nmem, size_t size)
  {
    void *w;
    if ((max_size_test/size) < nmem) {
      mp_fputs("Memory␣size␣overflow!\n", mp→err_out);
      mp→history = mp_fatal_error_stop;
      mp_jump_out(mp);
    }
    w = realloc(p, (nmem * size));
    if (w ≡ Λ) {
      mp_fputs("Out␣of␣memory!\n", mp→err_out);
      mp→history = mp_system_error_stop;
      mp_jump_out(mp);
    }
    return w;
  }
  void *mp_xmalloc(MP mp, size_t nmem, size_t size)
  {
    void *w;
#if DEBUG
    if ((max_size_test/size) < nmem) {
      mp_fputs("Memory␣size␣overflow!\n", mp→err_out);
      mp→history = mp_fatal_error_stop;
      mp_jump_out(mp);
    }
#endif
    w = malloc(nmem * size);
    if (w ≡ Λ) {
      mp_fputs("Out␣of␣memory!\n", mp→err_out);
      mp→history = mp_system_error_stop;
      mp_jump_out(mp);
    }
    return w;
  }
```

172.   ⟨Internal library declarations 10⟩ +≡

#**define** *mp_snprintf*(**void**)   *snprintf*

173. Dynamic memory allocation.

The METAPOST system does nearly all of its own memory allocation, so that it can readily be transported into environments that do not have automatic facilities for strings, garbage collection, etc., and so that it can be in control of what error messages the user receives.

#define MP_VOID (**mp_node**)(1)   /∗ Λ + 1, a Λ pointer different from Λ ∗/
#define *mp_link*(A) (A)→*link*   /∗ the *link* field of a node ∗/
#define *set_mp_link*(A, B) **do**
    {
      mp_node $d = (B)$;
        /∗ *printf*("set␣link␣␣␣␣of␣%p␣to␣%p␣on␣line␣%d\n", (A), d, __LINE__); ∗/
      mp_link((A)) = d;
    }
    while (0)
#define *mp_type*(A) (A)→*type*   /∗ identifies what kind of value this is ∗/
#define *mp_name_type*(A) (A)→*name_type*   /∗ a clue to the name of this value ∗/

174. ⟨ MPlib internal header stuff 6 ⟩ +≡
#define NODE_BODY *mp_variable_type type*;
  mp_name_type_type name_type;

  unsigned short *has_number*; **struct** *mp_node_data* ∗*link*
  typedef struct mp_node_data {
    NODE_BODY;

    mp_value_data *data*;
  } **mp_node_data**;
  typedef struct mp_node_data ∗**mp_symbolic_node**;

175. Users who wish to study the memory requirements of particular applications can can use the special features that keep track of current and maximum memory usage. METAPOST will report these statistics when *mp_tracing_stats* is positive.

#define *add_var_used*(a) **do**
    {
      mp→*var_used* += (a);
      if (*mp*→*var_used* > *mp*→*var_used_max*) *mp*→*var_used_max* = *mp*→*var_used*;
    }
    while (0)
⟨ Global variables 14 ⟩ +≡
  size_t *var_used*;   /∗ how much memory is in use ∗/
  size_t *var_used_max*;   /∗ how much memory was in use max ∗/

176.   These redirect to function to aid in debugging.

#**if** DEBUG
#**define** $mp\_sym\_info(A)get\_mp\_sym\_info$ $(mp,(A))$
#**define** $set\_mp\_sym\_info(A,B)do\_set\_mp\_sym\_info$ $(mp,(A),(B))$
#**define** $mp\_sym\_sym(A)get\_mp\_sym\_sym$ $(mp,(A))$
#**define** $set\_mp\_sym\_sym(A,B)do\_set\_mp\_sym\_sym$ $(mp,(A),(\mathbf{mp\_sym})(B))$
  static void $do\_set\_mp\_sym\_info(\mathbf{MP}\ mp, \mathbf{mp\_node}\ p, \mathbf{halfword}\ v)$
  {
    FUNCTION_TRACE3("do_set_mp_sym_info(%p,%d)\n", p, v);
    $assert(p\rightarrow type \equiv mp\_symbol\_node)$;
    $set\_indep\_value(p, v)$;
  }
  static halfword $get\_mp\_sym\_info(\mathbf{MP}\ mp, \mathbf{mp\_node}\ p)$
  {
    FUNCTION_TRACE3("%d␣=␣get_mp_sym_info(%p)\n", $indep\_value(p), p$);
    $assert(p\rightarrow type \equiv mp\_symbol\_node)$;
    return $indep\_value(p)$;
  }
  static void $do\_set\_mp\_sym\_sym(\mathbf{MP}\ mp, \mathbf{mp\_node}\ p, \mathbf{mp\_sym}\ v)$
  {
    $\mathbf{mp\_symbolic\_node}\ pp = (\mathbf{mp\_symbolic\_node})\ p$;
    FUNCTION_TRACE3("do_set_mp_sym_sym(%p,%p)\n", pp, v);
    $assert(pp\rightarrow type \equiv mp\_symbol\_node)$;
    $pp\rightarrow data.sym = v$;
  }
  static mp_sym $get\_mp\_sym\_sym(\mathbf{MP}\ mp, \mathbf{mp\_node}\ p)$
  {
    $\mathbf{mp\_symbolic\_node}\ pp = (\mathbf{mp\_symbolic\_node})\ p$;
    FUNCTION_TRACE3("%p␣=␣get_mp_sym_sym(%p)\n", $pp\rightarrow data.sym, pp$);
    $assert(pp\rightarrow type \equiv mp\_symbol\_node)$;
    return $pp\rightarrow data.sym$;
  }
#**else**
#**define** $mp\_sym\_info(A)indep\_value$ (A)
#**define** $set\_mp\_sym\_info(A,B)set\_indep\_value$ $(A,(B))$
#**define** $mp\_sym\_sym(A)(A)\rightarrow data.sym$
#**define** $set\_mp\_sym\_sym(A,B)(A)\rightarrow data.sym = (\mathbf{mp\_sym})(B)$
#**endif**

177.   ⟨Declarations 8⟩ +≡
#**if** DEBUG
  static void $do\_set\_mp\_sym\_info(\mathbf{MP}\ mp, \mathbf{mp\_node}\ A, \mathbf{halfword}\ B)$;
  static halfword $get\_mp\_sym\_info(\mathbf{MP}\ mp, \mathbf{mp\_node}\ p)$;
  static void $do\_set\_mp\_sym\_sym(\mathbf{MP}\ mp, \mathbf{mp\_node}\ A, \mathbf{mp\_sym}\ B)$;
  static mp_sym $get\_mp\_sym\_sym(\mathbf{MP}\ mp, \mathbf{mp\_node}\ p)$;
#**endif**

178.   The function *get_symbolic_node* returns a pointer to a new symbolic node whose *link* field is null.

#define *symbolic_node_size* **sizeof**(**mp_node_data**)

```
static mp_node mp_get_symbolic_node(MP mp)
{
    mp_symbolic_node p;
    if (mp→symbolic_nodes) {
        p = (mp_symbolic_node) mp→symbolic_nodes;
        mp→symbolic_nodes = p→link;
        mp→num_symbolic_nodes −−;
        p→link = Λ;
    }
    else {
        p = malloc_node(symbolic_node_size);
        new_number(p→data.n);
        p→has_number = 1;
    }
    p→type = mp_symbol_node;
    p→name_type = mp_normal_sym;
    FUNCTION_TRACE2("%p=_mp_get_symbolic_node()\n", p);
    return (mp_node) p;
}
```

179.   Conversely, when some node p of size s is no longer needed, the operation $\textit{free\_node}\,(p, s)$ will make its words available, by inserting p as a new empty node just before where $\textit{rover}$ now points.

A symbolic node is recycled by calling $\textit{free\_symbolic\_node}$.

void $\textit{mp\_free\_node}\,(\textbf{MP}\ \textit{mp}, \textbf{mp\_node}\ p, \textbf{size\_t}\ \textit{siz}\,)$
$\{$ /∗ node liberation ∗/
  FUNCTION_TRACE3("mp_free_node(%p,%d)\n", p, (**int**) $\textit{siz}$);
  if ($\neg p$) **return**;
  $\textit{mp}{\rightarrow}\textit{var\_used} \mathrel{-=} \textit{siz}$;
  if ($\textit{mp}{\rightarrow}\textit{math\_mode} > \textit{mp\_math\_double\_mode}$) $\{$
    if ($p{\rightarrow}\textit{has\_number} \geq 1 \wedge \textit{is\_number}\,(((\textbf{mp\_symbolic\_node})\ p){\rightarrow}\textit{data}.n))$ $\{$
      $\textit{free\_number}\,(((\textbf{mp\_symbolic\_node})\ p){\rightarrow}\textit{data}.n)$;
    $\}$
    if ($p{\rightarrow}\textit{has\_number} \equiv 2 \wedge \textit{is\_number}\,(((\textbf{mp\_value\_node})\ p){\rightarrow}\textit{subscript\_}))$ $\{$
      $\textit{free\_number}\,(((\textbf{mp\_value\_node})\ p){\rightarrow}\textit{subscript\_})$;
    $\}$ /∗ There was a quite large **switch** here first, but the $\textit{mp\_dash\_node}$ case was the only one that
        did anything ... ∗/
    if ($\textit{mp\_type}\,(p) \equiv \textit{mp\_dash\_node\_type}$) $\{$
      $\textit{free\_number}\,(((\textit{mp\_dash\_node})p){\rightarrow}\textit{start\_x})$;
      $\textit{free\_number}\,(((\textit{mp\_dash\_node})p){\rightarrow}\textit{stop\_x})$;
      $\textit{free\_number}\,(((\textit{mp\_dash\_node})p){\rightarrow}\textit{dash\_y})$;
    $\}$
  $\}$
  $\textit{xfree}\,(p)$;
$\}$

void $\textit{mp\_free\_symbolic\_node}\,(\textbf{MP}\ \textit{mp}, \textbf{mp\_node}\ p)$
$\{$ /∗ node liberation ∗/
  FUNCTION_TRACE2("mp_free_symbolic_node(%p)\n", p);
  if ($\neg p$) **return**;
  if ($\textit{mp}{\rightarrow}\textit{num\_symbolic\_nodes} < \textit{max\_num\_symbolic\_nodes}$) $\{$
    $p{\rightarrow}\textit{link} = \textit{mp}{\rightarrow}\textit{symbolic\_nodes}$;
    $\textit{mp}{\rightarrow}\textit{symbolic\_nodes} = p$;
    $\textit{mp}{\rightarrow}\textit{num\_symbolic\_nodes}{+}{+}$;
    return;
  $\}$
  $\textit{mp}{\rightarrow}\textit{var\_used} \mathrel{-=} \textit{symbolic\_node\_size}$;
  $\textit{xfree}\,(p)$;
$\}$

void $\textit{mp\_free\_value\_node}\,(\textbf{MP}\ \textit{mp}, \textbf{mp\_node}\ p)$
$\{$ /∗ node liberation ∗/
  FUNCTION_TRACE2("mp_free_value_node(%p)\n", p);
  if ($\neg p$) **return**;
  if ($\textit{mp}{\rightarrow}\textit{num\_value\_nodes} < \textit{max\_num\_value\_nodes}$) $\{$
    $p{\rightarrow}\textit{link} = \textit{mp}{\rightarrow}\textit{value\_nodes}$;
    $\textit{mp}{\rightarrow}\textit{value\_nodes} = p$;
    $\textit{mp}{\rightarrow}\textit{num\_value\_nodes}{+}{+}$;
    return;
  $\}$
  $\textit{mp}{\rightarrow}\textit{var\_used} \mathrel{-=} \textit{value\_node\_size}$;
  $\textit{assert}\,(p{\rightarrow}\textit{has\_number} \equiv 2)$;
  if ($\textit{mp}{\rightarrow}\textit{math\_mode} > \textit{mp\_math\_double\_mode}$) $\{$
    $\textit{free\_number}\,(((\textbf{mp\_value\_node})\ p){\rightarrow}\textit{data}.n)$;

$free\_number(((\textbf{mp\_value\_node})\ p){\rightarrow}subscript\_);$
   }
   $xfree(p);$
 }

180.   ⟨Internal library declarations 10⟩ +≡
   void $mp\_free\_node(\textbf{MP}\ mp, \textbf{mp\_node}\ p, \textbf{size\_t}\ siz);$
   void $mp\_free\_symbolic\_node(\textbf{MP}\ mp, \textbf{mp\_node}\ p);$
   void $mp\_free\_value\_node(\textbf{MP}\ mp, \textbf{mp\_node}\ p);$

181.   Memory layout.   Some nodes are created statically, since static allocation is more efficient than dynamic allocation when we can get away with it.

⟨ Global variables 14 ⟩ +≡
  mp_dash_node null_dash ;

  mp_value_node *dep_head* ;
  mp_node *inf_val* ;
  mp_node *zero_val* ;
  mp_node *temp_val* ;
  mp_node *end_attr* ;
  mp_node *bad_vardef* ;
  mp_node *temp_head* ;
  mp_node *hold_head* ;
  mp_node *spec_head* ;

182.   The following code gets the memory off to a good start.

⟨ Initialize table entries 182 ⟩ ≡
  mp→spec_head = *mp_get_symbolic_node* (*mp*);
  mp→last_pending = *mp→spec_head* ;
  mp→temp_head = *mp_get_symbolic_node* (*mp*);
  mp→hold_head = *mp_get_symbolic_node* (*mp*);
See also sections 202, 203, 226, 227, 258, 368, 385, 448, 479, 611, 615, 628, 668, 763, 830, 927, 970, 1000, 1193, 1198, 1207, and 1212.

This code is used in section 1297.

183.   ⟨ Free table entries 183 ⟩ ≡
  mp_free_symbolic_node (*mp* , *mp→spec_head*);
  mp_free_symbolic_node (*mp* , *mp→temp_head*);
  mp_free_symbolic_node (*mp* , *mp→hold_head*);
See also sections 259, 480, 629, 669, 764, 901, 971, 1001, 1194, and 1208.

This code is used in section 12.

184.   The procedure *flush_node_list* (*p*) frees an entire linked list of nodes that starts at a given position, until coming to a Λ pointer.

```
static void mp_flush_node_list(MP mp, mp_node p)
{
  mp_node q;      /* the node being recycled */
  FUNCTION_TRACE2("mp_flush_node_list(%p)\n", p);
  while (p ≠ Λ) {
    q = p;
    p = p→link;
    if (q→type ≠ mp_symbol_node) mp_free_token_node(mp, q);
    else mp_free_symbolic_node(mp, q);
  }
}
```

185.   The command codes.   Before we can go much further, we need to define symbolic names for the internal code numbers that represent the various commands obeyed by METAPOST. These codes are somewhat arbitrary, but not completely so. For example, some codes have been made adjacent so that **case** statements in the program need not consider cases that are widely spaced, or so that **case** statements can be replaced by **if** statements. A command can begin an expression if and only if its code lies between *min_primary_command* and *max_primary_command*, inclusive. The first token of a statement that doesn't begin with an expression has a command code between *min_command* and *max_statement_command*, inclusive. Anything less than *min_command* is eliminated during macro expansions, and anything no more than *max_pre_command* is eliminated when expanding TeX material. Ranges such as *min_secondary_command* .. *max_secondary_command* are used when parsing expressions, but the relative ordering within such a range is generally not critical.

The ordering of the highest-numbered commands (*comma* < *semicolon* < *end_group* < *stop*) is crucial for the parsing and error-recovery methods of this program as is the ordering *if_test* < *fi_or_else* for the smallest two commands. The ordering is also important in the ranges *numeric_token* .. *plus_or_minus* and *left_brace* .. *ampersand*.

At any rate, here is the list, for future reference.

#define *mp_max_command_code*   *mp_stop*
#define *mp_max_pre_command*   *mp_mpx_break*
#define *mp_min_command*   (*mp_defined_macro* + 1)
#define *mp_max_statement_command*   *mp_type_name*
#define *mp_min_primary_command*   *mp_type_name*
#define *mp_min_suffix_token*   *mp_internal_quantity*
#define *mp_max_suffix_token*   *mp_numeric_token*
#define *mp_max_primary_command*   *mp_plus_or_minus*   /∗ should also be *numeric_token* + 1 ∗/
#define *mp_min_tertiary_command*   *mp_plus_or_minus*
#define *mp_max_tertiary_command*   *mp_tertiary_binary*
#define *mp_min_expression_command*   *mp_left_brace*
#define *mp_max_expression_command*   *mp_equals*
#define *mp_min_secondary_command*   *mp_and_command*
#define *mp_max_secondary_command*   *mp_secondary_binary*
#define *mp_end_of_statement*   (*cur_cmd*( ) > *mp_comma*)

⟨ Enumeration types 185 ⟩ ≡
  typedef enum { *mp_start_tex* = 1,   /∗ begin TeX material (**btex**, **verbatimtex**) ∗/
  mp_etex_marker,   /∗ end TeX material (**etex**) ∗/
  mp_mpx_break,   /∗ stop reading an MPX file (**mpxbreak**) ∗/
  mp_if_test,   /∗ conditional text (**if**) ∗/
  mp_fi_or_else,   /∗ delimiters for conditionals (**elseif**, **else**, **fi**) ∗/
  mp_input,   /∗ input a source file (**input**, **endinput**) ∗/
  mp_iteration,   /∗ iterate (**for**, **forsuffixes**, **forever**, **endfor**) ∗/
  mp_repeat_loop,   /∗ special command substituted for **endfor** ∗/
  mp_exit_test,   /∗ premature exit from a loop (**exitif**) ∗/
  mp_relax,   /∗ do nothing (\) ∗/
  mp_scan_tokens,   /∗ put a string into the input buffer ∗/
  mp_expand_after,   /∗ look ahead one token ∗/
  mp_defined_macro,   /∗ a macro defined by the user ∗/
  mp_save_command,   /∗ save a list of tokens (**save**) ∗/
  mp_interim_command,   /∗ save an internal quantity (**interim**) ∗/
  mp_let_command,   /∗ redefine a symbolic token (**let**) ∗/
  mp_new_internal,   /∗ define a new internal quantity (**newinternal**) ∗/
  mp_macro_def,   /∗ define a macro (**def**, **vardef**, etc.) ∗/
  mp_ship_out_command,   /∗ output a character (**shipout**) ∗/
  mp_add_to_command,   /∗ add to edges (**addto**) ∗/

mp_bounds_command,   /∗ add bounding path to edges (**setbounds**, **clip**) ∗/
mp_tfm_command,   /∗ command for font metric info (**ligtable**, etc.) ∗/
mp_protection_command,   /∗ set protection flag (**outer**, **inner**) ∗/
mp_show_command,   /∗ diagnostic output (**show**, **showvariable**, etc.) ∗/
mp_mode_command,   /∗ set interaction level (**batchmode**, etc.) ∗/
mp_random_seed,   /∗ initialize random number generator (**randomseed**) ∗/
mp_message_command,   /∗ communicate to user (**message**, **errmessage**) ∗/
mp_every_job_command,   /∗ designate a starting token (**everyjob**) ∗/
mp_delimiters,   /∗ define a pair of delimiters (**delimiters**) ∗/
mp_special_command,
   /∗ output special info (**special**) or font map info (**fontmapfile**, **fontmapline**) ∗/
mp_write_command,   /∗ write text to a file (**write**) ∗/
mp_type_name,   /∗ declare a type (**numeric**, **pair**, etc.) ∗/
mp_left_delimiter,   /∗ the left delimiter of a matching pair ∗/
mp_begin_group,   /∗ beginning of a group (**begingroup**) ∗/
mp_nullary,   /∗ an operator without arguments (e.g., **normaldeviate**) ∗/
mp_unary,   /∗ an operator with one argument (e.g., **sqrt**) ∗/
mp_str_op,   /∗ convert a suffix to a string (**str**) ∗/
mp_cycle,   /∗ close a cyclic path (**cycle**) ∗/
mp_primary_binary,   /∗ binary operation taking 'of' (e.g., **point**) ∗/
mp_capsule_token,   /∗ a value that has been put into a token list ∗/
mp_string_token,   /∗ a string constant (e.g., `"hello"`) ∗/
mp_internal_quantity,   /∗ internal numeric parameter (e.g., **pausing**) ∗/
mp_tag_token,   /∗ a symbolic token without a primitive meaning ∗/
mp_numeric_token,   /∗ a numeric constant (e.g., `3.14159`) ∗/
mp_plus_or_minus,   /∗ either '+' or '−' ∗/
mp_tertiary_secondary_macro,   /∗ a macro defined by **secondarydef** ∗/
mp_tertiary_binary,   /∗ an operator at the tertiary level (e.g., '++') ∗/
mp_left_brace,   /∗ the operator '{' ∗/
mp_path_join,   /∗ the operator '..' ∗/
mp_ampersand,   /∗ the operator '&' ∗/
mp_expression_tertiary_macro,   /∗ a macro defined by **tertiarydef** ∗/
mp_expression_binary,   /∗ an operator at the expression level (e.g., '<') ∗/
mp_equals,   /∗ the operator '=' ∗/
mp_and_command,   /∗ the operator '**and**' ∗/
mp_secondary_primary_macro,   /∗ a macro defined by **primarydef** ∗/
mp_slash,   /∗ the operator '/' ∗/
mp_secondary_binary,   /∗ an operator at the binary level (e.g., **shifted**) ∗/
mp_param_type,   /∗ type of parameter (**primary**, **expr**, **suffix**, etc.) ∗/
mp_controls,   /∗ specify control points explicitly (**controls**) ∗/
mp_tension,   /∗ specify tension between knots (**tension**) ∗/
mp_at_least,   /∗ bounded tension value (**atleast**) ∗/
mp_curl_command,   /∗ specify curl at an end knot (**curl**) ∗/
mp_macro_special,   /∗ special macro operators (**quote**, `#@!`, etc.) ∗/
mp_right_delimiter,   /∗ the right delimiter of a matching pair ∗/
mp_left_bracket,   /∗ the operator '[' ∗/
mp_right_bracket,   /∗ the operator ']' ∗/
mp_right_brace,   /∗ the operator '}' ∗/
mp_with_option,   /∗ option for filling (**withpen**, **withweight**, etc.) ∗/
mp_thing_to_add,   /∗ variant of **addto** (**contour**, **doublepath**, **also**) ∗/
mp_of_token,   /∗ the operator '**of**' ∗/
mp_to_token,   /∗ the operator '**to**' ∗/

mp_step_token,   /∗ the operator '**step**' ∗/
mp_until_token,   /∗ the operator '**until**' ∗/
mp_within_token,   /∗ the operator '**within**' ∗/
$mp\_lig\_kern\_token$,   /∗ the operators '**kern**' and '**=:**' and '**=:|**', etc. ∗/
$mp\_assignment$,   /∗ the operator ':**=**' ∗/
mp_skip_to,   /∗ the operation '**skipto**' ∗/
mp_bchar_label,   /∗ the operator '**||:**' ∗/
mp_double_colon,   /∗ the operator '**::**' ∗/
mp_colon,   /∗ the operator '**:**' ∗/

mp_comma,   /∗ the operator '**,**', must be $colon + 1$ ∗/
$mp\_semicolon$,   /∗ the operator '**;**', must be $comma + 1$ ∗/
mp_end_group,   /∗ end a group (**endgroup**), must be $semicolon + 1$ ∗/
mp_stop,   /∗ end a job (**end**, **dump**), must be $end\_group + 1$ ∗/
mp_outer_tag,   /∗ protection code added to command code ∗/
$mp\_undefined\_cs$,   /∗ protection code added to command code ∗/
} $mp\_command\_code$;

See also sections 186 and 189.

This code is used in section 4.

186.   Variables and capsules in METAPOST have a variety of "types," distinguished by the code numbers defined here. These numbers are also not completely arbitrary. Things that get expanded must have types $> mp\_independent$; a type remaining after expansion is numeric if and only if its code number is at least $numeric\_type$; objects containing numeric parts must have types between $transform\_type$ and $pair\_type$; all other types must be smaller than $transform\_type$; and among the types that are not unknown or vacuous, the smallest two must be $boolean\_type$ and $string\_type$ in that order.

#define $unknown\_tag$ 1   /* this constant is added to certain type codes below */
#define $unknown\_types$ $mp\_unknown\_boolean$: **case** $mp\_unknown\_string$: **case** $mp\_unknown\_pen$:
  case $mp\_unknown\_picture$: **case** $mp\_unknown\_path$

⟨ Enumeration types 185 ⟩ +≡
 typedef enum { $mp\_undefined = 0$,   /* no type has been declared */
 $mp\_vacuous$,   /* no expression was present */
 $mp\_boolean\_type$,   /* **boolean** with a known value */
 $mp\_unknown\_boolean$, mp_string_type,   /* **string** with a known value */
 $mp\_unknown\_string$, mp_pen_type,   /* **pen** with a known value */
 $mp\_unknown\_pen$, mp_path_type,   /* **path** with a known value */
 $mp\_unknown\_path$, $mp\_picture\_type$,   /* **picture** with a known value */
 $mp\_unknown\_picture$, $mp\_transform\_type$,   /* **transform** variable or capsule */
 mp_color_type,   /* **color** variable or capsule */
 $mp\_cmykcolor\_type$,   /* **cmykcolor** variable or capsule */
 mp_pair_type,   /* **pair** variable or capsule */
 $mp\_numeric\_type$,   /* variable that has been declared **numeric** but not used */
 mp_known,   /* **numeric** with a known value */
 $mp\_dependent$,   /* a linear combination with $fraction$ coefficients */
 $mp\_proto\_dependent$,   /* a linear combination with $scaled$ coefficients */
 $mp\_independent$,   /* **numeric** with unknown value */
 mp_token_list,   /* variable name or suffix argument or text argument */
 $mp\_structured$,   /* variable with subscripts and attributes */
 $mp\_unsuffixed\_macro$,   /* variable defined with **vardef** but no @!# */
 $mp\_suffixed\_macro$,   /* variable defined with **vardef** and @!# */
  /* here are some generic node types */
 mp_symbol_node, $mp\_token\_node\_type$, $mp\_value\_node\_type$, $mp\_attr\_node\_type$, $mp\_subscr\_node\_type$,
  $mp\_pair\_node\_type$, $mp\_transform\_node\_type$, $mp\_color\_node\_type$, $mp\_cmykcolor\_node\_type$,
  /* it is important that the next 7 items remain in this order, for export */
 $mp\_fill\_node\_type$, $mp\_stroked\_node\_type$, $mp\_text\_node\_type$, $mp\_start\_clip\_node\_type$,
  $mp\_start\_bounds\_node\_type$, $mp\_stop\_clip\_node\_type$, $mp\_stop\_bounds\_node\_type$, $mp\_dash\_node\_type$,
  $mp\_dep\_node\_type$, $mp\_if\_node\_type$, $mp\_edge\_header\_node\_type$, } $mp\_variable\_type$;

187.   ⟨ Declarations 8 ⟩ +≡
 static void mp_print_type(**MP** mp, **quarterword** t);

188.   ⟨Basic printing procedures 85⟩ +≡
  static const char *mp_type_string(**quarterword** t)
  {
    const char *$s = \Lambda$;

    switch (t) {
    case $mp\_undefined$: $s =$ "undefined";
      break;
    case $mp\_vacuous$: $s =$ "vacuous";
      break;
    case $mp\_boolean\_type$: $s =$ "boolean";
      break;
    case $mp\_unknown\_boolean$: $s =$ "unknown␣boolean";
      break;
    case mp_string_type: $s =$ "string";
      break;
    case $mp\_unknown\_string$: $s =$ "unknown␣string";
      break;
    case mp_pen_type: $s =$ "pen";
      break;
    case $mp\_unknown\_pen$: $s =$ "unknown␣pen";
      break;
    case mp_path_type: $s =$ "path";
      break;
    case $mp\_unknown\_path$: $s =$ "unknown␣path";
      break;
    case $mp\_picture\_type$: $s =$ "picture";
      break;
    case $mp\_unknown\_picture$: $s =$ "unknown␣picture";
      break;
    case $mp\_transform\_type$: $s =$ "transform";
      break;
    case mp_color_type: $s =$ "color";
      break;
    case $mp\_cmykcolor\_type$: $s =$ "cmykcolor";
      break;
    case mp_pair_type: $s =$ "pair";
      break;
    case mp_known: $s =$ "known␣numeric";
      break;
    case $mp\_dependent$: $s =$ "dependent";
      break;
    case $mp\_proto\_dependent$: $s =$ "proto-dependent";
      break;
    case $mp\_numeric\_type$: $s =$ "numeric";
      break;
    case $mp\_independent$: $s =$ "independent";
      break;
    case mp_token_list: $s =$ "token␣list";
      break;
    case $mp\_structured$: $s =$ "mp_structured";
      break;
    case $mp\_unsuffixed\_macro$: $s =$ "unsuffixed␣macro";

```
    break;
case mp_suffixed_macro: s = "suffixed␣macro";
    break;
case mp_symbol_node: s = "symbol␣node";
    break;
case mp_token_node_type: s = "token␣node";
    break;
case mp_value_node_type: s = "value␣node";
    break;
case mp_attr_node_type: s = "attribute␣node";
    break;
case mp_subscr_node_type: s = "subscript␣node";
    break;
case mp_pair_node_type: s = "pair␣node";
    break;
case mp_transform_node_type: s = "transform␣node";
    break;
case mp_color_node_type: s = "color␣node";
    break;
case mp_cmykcolor_node_type: s = "cmykcolor␣node";
    break;
case mp_fill_node_type: s = "fill␣node";
    break;
case mp_stroked_node_type: s = "stroked␣node";
    break;
case mp_text_node_type: s = "text␣node";
    break;
case mp_start_clip_node_type: s = "start␣clip␣node";
    break;
case mp_start_bounds_node_type: s = "start␣bounds␣node";
    break;
case mp_stop_clip_node_type: s = "stop␣clip␣node";
    break;
case mp_stop_bounds_node_type: s = "stop␣bounds␣node";
    break;
case mp_dash_node_type: s = "dash␣node";
    break;
case mp_dep_node_type: s = "dependency␣node";
    break;
case mp_if_node_type: s = "if␣node";
    break;
case mp_edge_header_node_type: s = "edge␣header␣node";
    break;
default:
    {
        char ss[256];
        mp_snprintf (ss, 256, "<unknown␣type␣%d>", t);
        s = strdup (ss);
    }
    break;
}
return s;
```

```
}
void mp_print_type (MP mp, quarterword t)
{
  if (t ≥ 0 ∧ t ≤ mp_edge_header_node_type)  mp_print (mp, mp_type_string (t));
  else  mp_print (mp, "unknown");
}
```

189.   Values inside METAPOST are stored in non-symbolic nodes that have a *name_type* as well as a *type*. The possibilities for *name_type* are defined here; they will be explained in more detail later.

⟨ Enumeration types 185 ⟩ +≡

  typedef enum {
    mp_root = 0,   /* *name_type* at the top level of a variable */
    mp_saved_root,   /* same, when the variable has been saved */
    mp_structured_root,   /* *name_type* where a *mp_structured* branch occurs */
    mp_subscr,   /* *name_type* in a subscript node */
    mp_attr,   /* *name_type* in an attribute node */
    mp_x_part_sector,   /* *name_type* in the **xpart** of a node */
    mp_y_part_sector,   /* *name_type* in the **ypart** of a node */
    mp_xx_part_sector,   /* *name_type* in the **xxpart** of a node */
    mp_xy_part_sector,   /* *name_type* in the **xypart** of a node */
    mp_yx_part_sector,   /* *name_type* in the **yxpart** of a node */
    mp_yy_part_sector,   /* *name_type* in the **yypart** of a node */
    mp_red_part_sector,   /* *name_type* in the **redpart** of a node */
    mp_green_part_sector,   /* *name_type* in the **greenpart** of a node */
    mp_blue_part_sector,   /* *name_type* in the **bluepart** of a node */
    mp_cyan_part_sector,   /* *name_type* in the **redpart** of a node */
    mp_magenta_part_sector,   /* *name_type* in the **greenpart** of a node */
    mp_yellow_part_sector,   /* *name_type* in the **bluepart** of a node */
    mp_black_part_sector,   /* *name_type* in the **greenpart** of a node */
    mp_grey_part_sector,   /* *name_type* in the **bluepart** of a node */
    mp_capsule,   /* *name_type* in stashed-away subexpressions */
    mp_token,   /* *name_type* in a numeric token or string token */
      /* Symbolic nodes also have *name_type*, which is a different enumeration */
    mp_normal_sym, *mp_internal_sym*,   /* for values of internals */
    mp_macro_sym,   /* for macro names */
    mp_expr_sym,   /* for macro parameters if type *expr* */
    mp_suffix_sym,   /* for macro parameters if type *suffix* */
    mp_text_sym,   /* for macro parameters if type *text* */
    ⟨ Operation codes 190 ⟩
  } **mp_name_type_type**;

190.   Primitive operations that produce values have a secondary identification code in addition to their command code; it's something like genera and species. For example, '∗' has the command code *primary_binary*,■ and its secondary identification is *times*. The secondary codes start such that they don't overlap with the type codes; some type codes (e.g., *mp_string_type*) are used as operators as well as type identifications. The relative values are not critical, except for *true_code* .. *false_code*, *or_op* .. *and_op*, and *filled_op* .. *bounded_op*. The restrictions are that *and_op* − *false_code* = *or_op* − *true_code*, that the ordering of *x_part* ... *blue_part* must match that of *x_part_sector* .. *mp_blue_part_sector*, and the ordering of *filled_op* .. *bounded_op* must match that of the code values they test for.

#define   *mp_min_of*   *mp_substring_of*

⟨ Operation codes 190 ⟩ ≡
  mp_true_code,   /∗ operation code for `true` ∗/
  mp_false_code,   /∗ operation code for `false` ∗/
  mp_null_picture_code,   /∗ operation code for `nullpicture` ∗/
  mp_null_pen_code,   /∗ operation code for `nullpen` ∗/
  mp_read_string_op,   /∗ operation code for `readstring` ∗/
  mp_pen_circle,   /∗ operation code for `pencircle` ∗/
  mp_normal_deviate,   /∗ operation code for `normaldeviate` ∗/
  mp_read_from_op,   /∗ operation code for `readfrom` ∗/
  mp_close_from_op,   /∗ operation code for `closefrom` ∗/
  mp_odd_op,   /∗ operation code for `odd` ∗/
  mp_known_op,   /∗ operation code for `known` ∗/
  mp_unknown_op,   /∗ operation code for `unknown` ∗/
  mp_not_op,   /∗ operation code for `not` ∗/
  mp_decimal,   /∗ operation code for `decimal` ∗/
  mp_reverse,   /∗ operation code for `reverse` ∗/
  mp_make_path_op,   /∗ operation code for `makepath` ∗/
  mp_make_pen_op,   /∗ operation code for `makepen` ∗/
  mp_oct_op,   /∗ operation code for `oct` ∗/
  mp_hex_op,   /∗ operation code for `hex` ∗/
  mp_ASCII_op,   /∗ operation code for `ASCII` ∗/
  mp_char_op,   /∗ operation code for `char` ∗/
  mp_length_op,   /∗ operation code for `length` ∗/
  mp_turning_op,   /∗ operation code for `turningnumber` ∗/
  mp_color_model_part,   /∗ operation code for `colormodel` ∗/
  mp_x_part,   /∗ operation code for `xpart` ∗/
  mp_y_part,   /∗ operation code for `ypart` ∗/
  mp_xx_part,   /∗ operation code for `xxpart` ∗/
  mp_xy_part,   /∗ operation code for `xypart` ∗/
  mp_yx_part,   /∗ operation code for `yxpart` ∗/
  mp_yy_part,   /∗ operation code for `yypart` ∗/
  mp_red_part,   /∗ operation code for `redpart` ∗/
  mp_green_part,   /∗ operation code for `greenpart` ∗/
  mp_blue_part,   /∗ operation code for `bluepart` ∗/
  mp_cyan_part,   /∗ operation code for `cyanpart` ∗/
  mp_magenta_part,   /∗ operation code for `magentapart` ∗/
  mp_yellow_part,   /∗ operation code for `yellowpart` ∗/
  mp_black_part,   /∗ operation code for `blackpart` ∗/
  mp_grey_part,   /∗ operation code for `greypart` ∗/
  mp_font_part,   /∗ operation code for `fontpart` ∗/
  mp_text_part,   /∗ operation code for `textpart` ∗/
  mp_path_part,   /∗ operation code for `pathpart` ∗/
  mp_pen_part,   /∗ operation code for `penpart` ∗/

mp_dash_part,   /∗ operation code for `dashpart` ∗/
mp_prescript_part,   /∗ operation code for `prescriptpart` ∗/
mp_postscript_part,   /∗ operation code for `postscriptpart` ∗/
mp_sqrt_op,   /∗ operation code for `sqrt` ∗/
mp_m_exp_op,   /∗ operation code for `mexp` ∗/
mp_m_log_op,   /∗ operation code for `mlog` ∗/
mp_sin_d_op,   /∗ operation code for `sind` ∗/
mp_cos_d_op,   /∗ operation code for `cosd` ∗/
mp_floor_op,   /∗ operation code for `floor` ∗/
mp_uniform_deviate,   /∗ operation code for `uniformdeviate` ∗/
mp_char_exists_op,   /∗ operation code for `charexists` ∗/
mp_font_size,   /∗ operation code for `fontsize` ∗/
mp_ll_corner_op,   /∗ operation code for `llcorner` ∗/
mp_lr_corner_op,   /∗ operation code for `lrcorner` ∗/
mp_ul_corner_op,   /∗ operation code for `ulcorner` ∗/
mp_ur_corner_op,   /∗ operation code for `urcorner` ∗/
mp_arc_length,   /∗ operation code for `arclength` ∗/
mp_angle_op,   /∗ operation code for `angle` ∗/
mp_cycle_op,   /∗ operation code for `cycle` ∗/
mp_filled_op,   /∗ operation code for `filled` ∗/
mp_stroked_op,   /∗ operation code for `stroked` ∗/
mp_textual_op,   /∗ operation code for `textual` ∗/
mp_clipped_op,   /∗ operation code for `clipped` ∗/
mp_bounded_op,   /∗ operation code for `bounded` ∗/
mp_plus,   /∗ operation code for `+` ∗/
mp_minus,   /∗ operation code for `-` ∗/
mp_times,   /∗ operation code for `*` ∗/
mp_over,   /∗ operation code for `/` ∗/
mp_pythag_add,   /∗ operation code for `++` ∗/
mp_pythag_sub,   /∗ operation code for `+-+` ∗/
mp_or_op,   /∗ operation code for `or` ∗/
mp_and_op,   /∗ operation code for `and` ∗/
mp_less_than,   /∗ operation code for `<` ∗/
mp_less_or_equal,   /∗ operation code for `<=` ∗/
mp_greater_than,   /∗ operation code for `>` ∗/
mp_greater_or_equal,   /∗ operation code for `>=` ∗/
mp_equal_to,   /∗ operation code for `=` ∗/
mp_unequal_to,   /∗ operation code for `<>` ∗/
mp_concatenate,   /∗ operation code for `&` ∗/
mp_rotated_by,   /∗ operation code for `rotated` ∗/
mp_slanted_by,   /∗ operation code for `slanted` ∗/
mp_scaled_by,   /∗ operation code for `scaled` ∗/
mp_shifted_by,   /∗ operation code for `shifted` ∗/
mp_transformed_by,   /∗ operation code for `transformed` ∗/
mp_x_scaled,   /∗ operation code for `xscaled` ∗/
mp_y_scaled,   /∗ operation code for `yscaled` ∗/
mp_z_scaled,   /∗ operation code for `zscaled` ∗/
mp_in_font,   /∗ operation code for `infont` ∗/
mp_intersect,   /∗ operation code for `intersectiontimes` ∗/
mp_double_dot,   /∗ operation code for improper `..` ∗/
mp_substring_of,   /∗ operation code for `substring` ∗/
mp_subpath_of,   /∗ operation code for `subpath` ∗/

$mp\_direction\_time\_of$,   /∗ operation code for `directiontime` ∗/
mp_point_of,   /∗ operation code for `point` ∗/
$mp\_precontrol\_of$,   /∗ operation code for `precontrol` ∗/
$mp\_postcontrol\_of$,   /∗ operation code for `postcontrol` ∗/
$mp\_pen\_offset\_of$,   /∗ operation code for `penoffset` ∗/
$mp\_arc\_time\_of$,   /∗ operation code for `arctime` ∗/
$mp\_version$,   /∗ operation code for `mpversion` ∗/
$mp\_envelope\_of$,   /∗ operation code for `envelope` ∗/
mp_glyph_infont,   /∗ operation code for `glyph` ∗/
mp_kern_flag /∗ operation code for `kern` ∗/

This code is used in section 189.

191.   **static void** *mp_print_op*(**MP** *mp*, **quarterword** *c*)
  {
    if (*c* ≤ *mp_numeric_type*) {
      mp_print_type(*mp*, *c*);
    }
    else {
      switch (*c*) {
      case *mp_true_code*: *mp_print*(*mp*, "true");
        break;
      case *mp_false_code*: *mp_print*(*mp*, "false");
        break;
      case *mp_null_picture_code*: *mp_print*(*mp*, "nullpicture");
        break;
      case *mp_null_pen_code*: *mp_print*(*mp*, "nullpen");
        break;
      case *mp_read_string_op*: *mp_print*(*mp*, "readstring");
        break;
      case *mp_pen_circle*: *mp_print*(*mp*, "pencircle");
        break;
      case *mp_normal_deviate*: *mp_print*(*mp*, "normaldeviate");
        break;
      case *mp_read_from_op*: *mp_print*(*mp*, "readfrom");
        break;
      case *mp_close_from_op*: *mp_print*(*mp*, "closefrom");
        break;
      case *mp_odd_op*: *mp_print*(*mp*, "odd");
        break;
      case *mp_known_op*: *mp_print*(*mp*, "known");
        break;
      case *mp_unknown_op*: *mp_print*(*mp*, "unknown");
        break;
      case *mp_not_op*: *mp_print*(*mp*, "not");
        break;
      case *mp_decimal*: *mp_print*(*mp*, "decimal");
        break;
      case *mp_reverse*: *mp_print*(*mp*, "reverse");
        break;
      case *mp_make_path_op*: *mp_print*(*mp*, "makepath");
        break;
      case *mp_make_pen_op*: *mp_print*(*mp*, "makepen");
        break;
      case *mp_oct_op*: *mp_print*(*mp*, "oct");
        break;
      case *mp_hex_op*: *mp_print*(*mp*, "hex");
        break;
      case *mp_ASCII_op*: *mp_print*(*mp*, "ASCII");
        break;
      case *mp_char_op*: *mp_print*(*mp*, "char");
        break;
      case *mp_length_op*: *mp_print*(*mp*, "length");
        break;
      case *mp_turning_op*: *mp_print*(*mp*, "turningnumber");

  break;

case mp_x_part: $mp\_print(mp, "xpart")$;

  break;

case mp_y_part: $mp\_print(mp, "ypart")$;

  break;

case mp_xx_part: $mp\_print(mp, "xxpart")$;

  break;

case mp_xy_part: $mp\_print(mp, "xypart")$;

  break;

case mp_yx_part: $mp\_print(mp, "yxpart")$;

  break;

case mp_yy_part: $mp\_print(mp, "yypart")$;

  break;

case mp_red_part: $mp\_print(mp, "redpart")$;

  break;

case mp_green_part: $mp\_print(mp, "greenpart")$;

  break;

case mp_blue_part: $mp\_print(mp, "bluepart")$;

  break;

case mp_cyan_part: $mp\_print(mp, "cyanpart")$;

  break;

case $mp\_magenta\_part$: $mp\_print(mp, "magentapart")$;

  break;

case mp_yellow_part: $mp\_print(mp, "yellowpart")$;

  break;

case mp_black_part: $mp\_print(mp, "blackpart")$;

  break;

case mp_grey_part: $mp\_print(mp, "greypart")$;

  break;

case $mp\_color\_model\_part$: $mp\_print(mp, "colormodel")$;

  break;

case mp_font_part: $mp\_print(mp, "fontpart")$;

  break;

case mp_text_part: $mp\_print(mp, "textpart")$;

  break;

case $mp\_prescript\_part$: $mp\_print(mp, "prescriptpart")$;

  break;

case $mp\_postscript\_part$: $mp\_print(mp, "postscriptpart")$;

  break;

case mp_path_part: $mp\_print(mp, "pathpart")$;

  break;

case mp_pen_part: $mp\_print(mp, "penpart")$;

  break;

case mp_dash_part: $mp\_print(mp, "dashpart")$;

  break;

case mp_sqrt_op: $mp\_print(mp, "sqrt")$;

  break;

case $mp\_m\_exp\_op$: $mp\_print(mp, "mexp")$;

  break;

case $mp\_m\_log\_op$: $mp\_print(mp, "mlog")$;

  break;

case $mp\_sin\_d\_op$: $mp\_print(mp, "sind")$;

```
    break;
  case mp_cos_d_op: mp_print(mp, "cosd");
    break;
  case mp_floor_op: mp_print(mp, "floor");
    break;
  case mp_uniform_deviate: mp_print(mp, "uniformdeviate");
    break;
  case mp_char_exists_op: mp_print(mp, "charexists");
    break;
  case mp_font_size: mp_print(mp, "fontsize");
    break;
  case mp_ll_corner_op: mp_print(mp, "llcorner");
    break;
  case mp_lr_corner_op: mp_print(mp, "lrcorner");
    break;
  case mp_ul_corner_op: mp_print(mp, "ulcorner");
    break;
  case mp_ur_corner_op: mp_print(mp, "urcorner");
    break;
  case mp_arc_length: mp_print(mp, "arclength");
    break;
  case mp_angle_op: mp_print(mp, "angle");
    break;
  case mp_cycle_op: mp_print(mp, "cycle");
    break;
  case mp_filled_op: mp_print(mp, "filled");
    break;
  case mp_stroked_op: mp_print(mp, "stroked");
    break;
  case mp_textual_op: mp_print(mp, "textual");
    break;
  case mp_clipped_op: mp_print(mp, "clipped");
    break;
  case mp_bounded_op: mp_print(mp, "bounded");
    break;
  case mp_plus: mp_print_char(mp, xord('+'));
    break;
  case mp_minus: mp_print_char(mp, xord('-'));
    break;
  case mp_times: mp_print_char(mp, xord('*'));
    break;
  case mp_over: mp_print_char(mp, xord('/'));
    break;
  case mp_pythag_add: mp_print(mp, "++");
    break;
  case mp_pythag_sub: mp_print(mp, "+-+");
    break;
  case mp_or_op: mp_print(mp, "or");
    break;
  case mp_and_op: mp_print(mp, "and");
    break;
  case mp_less_than: mp_print_char(mp, xord('<'));
```

```
    break;
case mp_less_or_equal: mp_print(mp, "<=");
    break;
case mp_greater_than: mp_print_char(mp, xord('>'));
    break;
case mp_greater_or_equal: mp_print(mp, ">=");
    break;
case mp_equal_to: mp_print_char(mp, xord('='));
    break;
case mp_unequal_to: mp_print(mp, "<>");
    break;
case mp_concatenate: mp_print(mp, "&");
    break;
case mp_rotated_by: mp_print(mp, "rotated");
    break;
case mp_slanted_by: mp_print(mp, "slanted");
    break;
case mp_scaled_by: mp_print(mp, "scaled");
    break;
case mp_shifted_by: mp_print(mp, "shifted");
    break;
case mp_transformed_by: mp_print(mp, "transformed");
    break;
case mp_x_scaled: mp_print(mp, "xscaled");
    break;
case mp_y_scaled: mp_print(mp, "yscaled");
    break;
case mp_z_scaled: mp_print(mp, "zscaled");
    break;
case mp_in_font: mp_print(mp, "infont");
    break;
case mp_intersect: mp_print(mp, "intersectiontimes");
    break;
case mp_substring_of: mp_print(mp, "substring");
    break;
case mp_subpath_of: mp_print(mp, "subpath");
    break;
case mp_direction_time_of: mp_print(mp, "directiontime");
    break;
case mp_point_of: mp_print(mp, "point");
    break;
case mp_precontrol_of: mp_print(mp, "precontrol");
    break;
case mp_postcontrol_of: mp_print(mp, "postcontrol");
    break;
case mp_pen_offset_of: mp_print(mp, "penoffset");
    break;
case mp_arc_time_of: mp_print(mp, "arctime");
    break;
case mp_version: mp_print(mp, "mpversion");
    break;
case mp_envelope_of: mp_print(mp, "envelope");
```

```
         break;
      case mp_glyph_infont: mp_print(mp, "glyph");
         break;
      default: mp_print(mp, "..");
         break;
      }
   }
}
```

192.   METAPOST also has a bunch of internal parameters that a user might want to fuss with. Every such parameter has an identifying code number, defined here.

⟨ Types in the outer block 33 ⟩ +≡

    enum mp_given_internal { *mp_output_template* = 1,   /∗ a string set up by **outputtemplate** ∗/
    mp_output_filename,   /∗ the output file name, accessible as **outputfilename** ∗/
    mp_output_format,   /∗ the output format set up by **outputformat** ∗/
    mp_output_format_options,   /∗ the output format options set up by **outputformatoptions** ∗/
    mp_number_system,   /∗ the number system as set up by **numbersystem** ∗/
    mp_number_precision,   /∗ the number system precision as set up by **numberprecision** ∗/
    mp_job_name,   /∗ the perceived jobname, as set up from the options stucture, the name of the input
        file, or by **jobname** ∗/
    mp_tracing_titles,   /∗ show titles online when they appear ∗/
    mp_tracing_equations,   /∗ show each variable when it becomes known ∗/
    mp_tracing_capsules,   /∗ show capsules too ∗/
    mp_tracing_choices,   /∗ show the control points chosen for paths ∗/
    mp_tracing_specs,   /∗ show path subdivision prior to filling with polygonal a pen ∗/
    mp_tracing_commands,   /∗ show commands and operations before they are performed ∗/
    mp_tracing_restores,   /∗ show when a variable or internal is restored ∗/
    mp_tracing_macros,   /∗ show macros before they are expanded ∗/
    mp_tracing_output,   /∗ show digitized edges as they are output ∗/
    mp_tracing_stats,   /∗ show memory usage at end of job ∗/
    mp_tracing_lost_chars,   /∗ show characters that aren't **infont** ∗/
    mp_tracing_online,   /∗ show long diagnostics on terminal and in the log file ∗/
    mp_year,   /∗ the current year (e.g., 1984) ∗/
    mp_month,   /∗ the current month (e.g., 3 ≡ March) ∗/
    mp_day,   /∗ the current day of the month ∗/
    mp_time,   /∗ the number of minutes past midnight when this job started ∗/
    mp_hour,   /∗ the number of hours past midnight when this job started ∗/
    mp_minute,   /∗ the number of minutes in that hour when this job started ∗/
    mp_char_code,   /∗ the number of the next character to be output ∗/
    mp_char_ext,   /∗ the extension code of the next character to be output ∗/
    mp_char_wd,   /∗ the width of the next character to be output ∗/
    mp_char_ht,   /∗ the height of the next character to be output ∗/
    mp_char_dp,   /∗ the depth of the next character to be output ∗/
    mp_char_ic,   /∗ the italic correction of the next character to be output ∗/
    mp_design_size,   /∗ the unit of measure used for *mp_char_wd* .. *mp_char_ic*, in points ∗/
    mp_pausing,   /∗ positive to display lines on the terminal before they are read ∗/
    mp_showstopping,   /∗ positive to stop after each **show** command ∗/
    mp_fontmaking,   /∗ positive if font metric output is to be produced ∗/
    mp_linejoin,   /∗ as in PostScript: 0 for mitered, 1 for round, 2 for beveled ∗/
    mp_linecap,   /∗ as in PostScript: 0 for butt, 1 for round, 2 for square ∗/
    mp_miterlimit,   /∗ controls miter length as in PostScript ∗/
    mp_warning_check,   /∗ controls error message when variable value is large ∗/
    mp_boundary_char,   /∗ the right boundary character for ligatures ∗/
    mp_prologues,   /∗ positive to output conforming PostScript using built-in fonts ∗/
    mp_true_corners,   /∗ positive to make **llcorner** etc. ignore **setbounds** ∗/
    mp_default_color_model,   /∗ the default color model for unspecified items ∗/
    mp_restore_clip_color, *mp_procset*,   /∗ wether or not create PostScript command shortcuts ∗/
    mp_hppp,   /∗ horizontal pixels per point (for png output) ∗/
    mp_vppp,   /∗ vertical pixels per point (for png output) ∗/
    mp_gtroffmode ,   /∗ whether the user specified −*troff* on the command line ∗/
    } ;

```
typedef struct {
  mp_value v;
  char *intname;
} mp_internal;
```

193. ⟨MPlib internal header stuff 6⟩ +≡
#**define** *internal_value*(A)*mp*→*internal*[(A)].v.data.n
#**define** *set_internal_from_number*(A, B) **do**
  {
    number_clone(*internal_value*((A)), (B));
  }
  while (0)
#**define** *internal_string*(A)(**mp_string**)*mp*→*internal*[(A)].v.data.str
#**define** *set_internal_string*(A, B)*mp*→*internal*[(A)].v.data.str = (B)
#**define** *internal_name*(A)*mp*→*internal*[(A)].intname
#**define** *set_internal_name*(A, B)*mp*→*internal*[(A)].intname = (B)
#**define** *internal_type*(A)(*mp_variable_type*)*mp*→*internal*[(A)].v.type
#**define** *set_internal_type*(A, B)*mp*→*internal*[(A)].v.type = (B)
#**define** *set_internal_from_cur_exp*(A) **do**
  {
    if (*internal_type*((A)) ≡ *mp_string_type*) {
      add_str_ref(*cur_exp_str*());
      set_internal_string((A), *cur_exp_str*());
    }
    else {
      set_internal_from_number((A), *cur_exp_value_number*());
    }
  }
  while (0)

194.

#**define** *max_given_internal*   *mp_gtroffmode*
⟨Global variables 14⟩ +≡
  mp_internal *internal*;   /∗ the values of internal quantities ∗/
  int *int_ptr*;   /∗ the maximum internal quantity defined so far ∗/
  int *max_internal*;   /∗ current maximum number of internal quantities ∗/

195. ⟨Option variables 26⟩ +≡
  int *troff_mode*;

196.   ⟨Allocate or initialize variables 28⟩ +≡
  $mp\text{→}max\_internal = 2 * max\_given\_internal$;
  $mp\text{→}internal = xmalloc((mp\text{→}max\_internal + 1), \mathbf{sizeof}(\mathbf{mp\_internal}))$;
  $memset(mp\text{→}internal, 0, (\mathbf{size\_t})(mp\text{→}max\_internal + 1) * \mathbf{sizeof}(\mathbf{mp\_internal}))$;
  {
    int i;
    for $(i = 1;\ i \leq mp\text{→}max\_internal;\ i\text{++})$ {
      $new\_number(mp\text{→}internal[i].v.data.n)$;
    }
    for $(i = 1;\ i \leq max\_given\_internal;\ i\text{++})$ {
      $set\_internal\_type(i, mp\_known)$;
    }
  }
  $set\_internal\_type(mp\_output\_format, mp\_string\_type)$;
  $set\_internal\_type(mp\_output\_filename, mp\_string\_type)$;
  $set\_internal\_type(mp\_output\_format\_options, mp\_string\_type)$;
  $set\_internal\_type(mp\_output\_template, mp\_string\_type)$;
  $set\_internal\_type(mp\_number\_system, mp\_string\_type)$;
  $set\_internal\_type(mp\_job\_name, mp\_string\_type)$;
  $mp\text{→}troff\_mode = (opt\text{→}troff\_mode > 0\ ?\ true : false)$;

197.   ⟨Exported function headers 18⟩ +≡
  int $mp\_troff\_mode(\mathbf{MP}\ mp)$;

198.   **int** $mp\_troff\_mode(\mathbf{MP}\ mp)$
  {
    return $mp\text{→}troff\_mode$;
  }

199.   ⟨Set initial values of key variables 38⟩ +≡
  $mp\text{→}int\_ptr = max\_given\_internal$;

200.   The symbolic names for internal quantities are put into METAPOST's hash table by using a routine
called *primitive*, which will be defined later. Let us enter them now, so that we don't have to list all those
names again anywhere else.

⟨ Put each of METAPOST's primitives into the hash table 200 ⟩ ≡
  mp_primitive(*mp*, "tracingtitles", *mp_internal_quantity*, *mp_tracing_titles*);
  ;
  mp_primitive(*mp*, "tracingequations", *mp_internal_quantity*, *mp_tracing_equations*);
  ;
  mp_primitive(*mp*, "tracingcapsules", *mp_internal_quantity*, *mp_tracing_capsules*);
  ;
  mp_primitive(*mp*, "tracingchoices", *mp_internal_quantity*, *mp_tracing_choices*);
  ;
  mp_primitive(*mp*, "tracingspecs", *mp_internal_quantity*, *mp_tracing_specs*);
  ;
  mp_primitive(*mp*, "tracingcommands", *mp_internal_quantity*, *mp_tracing_commands*);
  ;
  mp_primitive(*mp*, "tracingrestores", *mp_internal_quantity*, *mp_tracing_restores*);
  ;
  mp_primitive(*mp*, "tracingmacros", *mp_internal_quantity*, *mp_tracing_macros*);
  ;
  mp_primitive(*mp*, "tracingoutput", *mp_internal_quantity*, *mp_tracing_output*);
  ;
  mp_primitive(*mp*, "tracingstats", *mp_internal_quantity*, *mp_tracing_stats*);
  ;
  mp_primitive(*mp*, "tracinglostchars", *mp_internal_quantity*, *mp_tracing_lost_chars*);
  ;
  mp_primitive(*mp*, "tracingonline", *mp_internal_quantity*, *mp_tracing_online*);
  ;
  mp_primitive(*mp*, "year", *mp_internal_quantity*, *mp_year*);
  ;
  mp_primitive(*mp*, "month", *mp_internal_quantity*, *mp_month*);
  ;
  mp_primitive(*mp*, "day", *mp_internal_quantity*, *mp_day*);
  ;
  mp_primitive(*mp*, "time", *mp_internal_quantity*, *mp_time*);
  ;
  mp_primitive(*mp*, "hour", *mp_internal_quantity*, *mp_hour*);
  ;
  mp_primitive(*mp*, "minute", *mp_internal_quantity*, *mp_minute*);
  ;
  mp_primitive(*mp*, "charcode", *mp_internal_quantity*, *mp_char_code*);
  ;
  mp_primitive(*mp*, "charext", *mp_internal_quantity*, *mp_char_ext*);
  ;
  mp_primitive(*mp*, "charwd", *mp_internal_quantity*, *mp_char_wd*);
  ;
  mp_primitive(*mp*, "charht", *mp_internal_quantity*, *mp_char_ht*);
  ;
  mp_primitive(*mp*, "chardp", *mp_internal_quantity*, *mp_char_dp*);
  ;
  mp_primitive(*mp*, "charic", *mp_internal_quantity*, *mp_char_ic*);
  ;

$mp\_primitive(mp, \texttt{"designsize"}, mp\_internal\_quantity, mp\_design\_size);$
;
$mp\_primitive(mp, \texttt{"pausing"}, mp\_internal\_quantity, mp\_pausing);$
;
$mp\_primitive(mp, \texttt{"showstopping"}, mp\_internal\_quantity, mp\_showstopping);$
;
$mp\_primitive(mp, \texttt{"fontmaking"}, mp\_internal\_quantity, mp\_fontmaking);$
;
$mp\_primitive(mp, \texttt{"linejoin"}, mp\_internal\_quantity, mp\_linejoin);$
;
$mp\_primitive(mp, \texttt{"linecap"}, mp\_internal\_quantity, mp\_linecap);$
;
$mp\_primitive(mp, \texttt{"miterlimit"}, mp\_internal\_quantity, mp\_miterlimit);$
;
$mp\_primitive(mp, \texttt{"warningcheck"}, mp\_internal\_quantity, mp\_warning\_check);$
;
$mp\_primitive(mp, \texttt{"boundarychar"}, mp\_internal\_quantity, mp\_boundary\_char);$
;
$mp\_primitive(mp, \texttt{"prologues"}, mp\_internal\_quantity, mp\_prologues);$
;
$mp\_primitive(mp, \texttt{"truecorners"}, mp\_internal\_quantity, mp\_true\_corners);$
;
$mp\_primitive(mp, \texttt{"mpprocset"}, mp\_internal\_quantity, mp\_procset);$
;
$mp\_primitive(mp, \texttt{"troffmode"}, mp\_internal\_quantity, mp\_gtroffmode);$
;
$mp\_primitive(mp, \texttt{"defaultcolormodel"}, mp\_internal\_quantity, mp\_default\_color\_model);$
;
$mp\_primitive(mp, \texttt{"restoreclipcolor"}, mp\_internal\_quantity, mp\_restore\_clip\_color);$
;
$mp\_primitive(mp, \texttt{"outputtemplate"}, mp\_internal\_quantity, mp\_output\_template);$
;
$mp\_primitive(mp, \texttt{"outputfilename"}, mp\_internal\_quantity, mp\_output\_filename);$
;
$mp\_primitive(mp, \texttt{"numbersystem"}, mp\_internal\_quantity, mp\_number\_system);$
;
$mp\_primitive(mp, \texttt{"numberprecision"}, mp\_internal\_quantity, mp\_number\_precision);$
;
$mp\_primitive(mp, \texttt{"outputformat"}, mp\_internal\_quantity, mp\_output\_format);$
;
$mp\_primitive(mp, \texttt{"outputformatoptions"}, mp\_internal\_quantity, mp\_output\_format\_options);$
;
$mp\_primitive(mp, \texttt{"jobname"}, mp\_internal\_quantity, mp\_job\_name);$
$mp\_primitive(mp, \texttt{"hppp"}, mp\_internal\_quantity, mp\_hppp);$
;
$mp\_primitive(mp, \texttt{"vppp"}, mp\_internal\_quantity, mp\_vppp);$
;

See also sections 232, 735, 745, 753, 759, 771, 809, 955, 1046, 1071, 1078, 1081, 1099, 1122, 1128, 1143, 1175, and 1185.

This code is used in section 1297.

201.   Colors can be specified in four color models. In the special case of *no_model*, MetaPost does not output any color operator to the postscript output.

Note: these values are passed directly on to *with_option*. This only works because the other possible values passed to *with_option* are 8 and 10 respectively (from *with_pen* and *with_picture*).

There is a first state, that is only used for *gs_colormodel*. It flags the fact that there has not been any kind of color specification by the user so far in the game.

⟨ MPlib header stuff 201 ⟩ ≡
  enum mp_color_model {
    mp_no_model = 1, *mp_grey_model* = 3, *mp_rgb_model* = 5, *mp_cmyk_model* = 7,
      mp_uninitialized_model = 9
  };

See also sections 299 and 457.

This code is used in section 3.

202.   ⟨ Initialize table entries 182 ⟩ +≡
  set_internal_from_number(*mp_default_color_model*, *unity_t*);
  number_multiply_int(*internal_value*(*mp_default_color_model*), *mp_rgb_model*);
  number_clone(*internal_value*(*mp_restore_clip_color*), *unity_t*);
  number_clone(*internal_value*(*mp_hppp*), *unity_t*);
  number_clone(*internal_value*(*mp_vppp*), *unity_t*);
  set_internal_string(*mp_output_template*, *mp_intern*(*mp*, `"%j.%c"`));
  set_internal_string(*mp_output_filename*, *mp_intern*(*mp*, `""`));
  set_internal_string(*mp_output_format*, *mp_intern*(*mp*, `"eps"`));
  set_internal_string(*mp_output_format_options*, *mp_intern*(*mp*, `""`));
  set_internal_string(*mp_number_system*, *mp_intern*(*mp*, `"scaled"`));
  set_internal_from_number(*mp_number_precision*, *precision_default*);
#**if** DEBUG
  number_clone(*internal_value*(*mp_tracing_titles*), *three_t*);
  number_clone(*internal_value*(*mp_tracing_equations*), *three_t*);
  number_clone(*internal_value*(*mp_tracing_capsules*), *three_t*);
  number_clone(*internal_value*(*mp_tracing_choices*), *three_t*);
  number_clone(*internal_value*(*mp_tracing_specs*), *three_t*);
  number_clone(*internal_value*(*mp_tracing_commands*), *three_t*);
  number_clone(*internal_value*(*mp_tracing_restores*), *three_t*);
  number_clone(*internal_value*(*mp_tracing_macros*), *three_t*);
  number_clone(*internal_value*(*mp_tracing_output*), *three_t*);
  number_clone(*internal_value*(*mp_tracing_stats*), *three_t*);
  number_clone(*internal_value*(*mp_tracing_lost_chars*), *three_t*);
  number_clone(*internal_value*(*mp_tracing_online*), *three_t*);
#**endif**

203.   Well, we do have to list the names one more time, for use in symbolic printouts.

⟨ Initialize table entries 182 ⟩ +≡
  set_internal_name (*mp_tracing_titles*, *xstrdup* ("tracingtitles"));
  set_internal_name (*mp_tracing_equations*, *xstrdup* ("tracingequations"));
  set_internal_name (*mp_tracing_capsules*, *xstrdup* ("tracingcapsules"));
  set_internal_name (*mp_tracing_choices*, *xstrdup* ("tracingchoices"));
  set_internal_name (*mp_tracing_specs*, *xstrdup* ("tracingspecs"));
  set_internal_name (*mp_tracing_commands*, *xstrdup* ("tracingcommands"));
  set_internal_name (*mp_tracing_restores*, *xstrdup* ("tracingrestores"));
  set_internal_name (*mp_tracing_macros*, *xstrdup* ("tracingmacros"));
  set_internal_name (*mp_tracing_output*, *xstrdup* ("tracingoutput"));
  set_internal_name (*mp_tracing_stats*, *xstrdup* ("tracingstats"));
  set_internal_name (*mp_tracing_lost_chars*, *xstrdup* ("tracinglostchars"));
  set_internal_name (*mp_tracing_online*, *xstrdup* ("tracingonline"));
  set_internal_name (*mp_year*, *xstrdup* ("year"));
  set_internal_name (*mp_month*, *xstrdup* ("month"));
  set_internal_name (*mp_day*, *xstrdup* ("day"));
  set_internal_name (*mp_time*, *xstrdup* ("time"));
  set_internal_name (*mp_hour*, *xstrdup* ("hour"));
  set_internal_name (*mp_minute*, *xstrdup* ("minute"));
  set_internal_name (*mp_char_code*, *xstrdup* ("charcode"));
  set_internal_name (*mp_char_ext*, *xstrdup* ("charext"));
  set_internal_name (*mp_char_wd*, *xstrdup* ("charwd"));
  set_internal_name (*mp_char_ht*, *xstrdup* ("charht"));
  set_internal_name (*mp_char_dp*, *xstrdup* ("chardp"));
  set_internal_name (*mp_char_ic*, *xstrdup* ("charic"));
  set_internal_name (*mp_design_size*, *xstrdup* ("designsize"));
  set_internal_name (*mp_pausing*, *xstrdup* ("pausing"));
  set_internal_name (*mp_showstopping*, *xstrdup* ("showstopping"));
  set_internal_name (*mp_fontmaking*, *xstrdup* ("fontmaking"));
  set_internal_name (*mp_linejoin*, *xstrdup* ("linejoin"));
  set_internal_name (*mp_linecap*, *xstrdup* ("linecap"));
  set_internal_name (*mp_miterlimit*, *xstrdup* ("miterlimit"));
  set_internal_name (*mp_warning_check*, *xstrdup* ("warningcheck"));
  set_internal_name (*mp_boundary_char*, *xstrdup* ("boundarychar"));
  set_internal_name (*mp_prologues*, *xstrdup* ("prologues"));
  set_internal_name (*mp_true_corners*, *xstrdup* ("truecorners"));
  set_internal_name (*mp_default_color_model*, *xstrdup* ("defaultcolormodel"));
  set_internal_name (*mp_procset*, *xstrdup* ("mpprocset"));
  set_internal_name (*mp_gtroffmode*, *xstrdup* ("troffmode"));
  set_internal_name (*mp_restore_clip_color*, *xstrdup* ("restoreclipcolor"));
  set_internal_name (*mp_output_template*, *xstrdup* ("outputtemplate"));
  set_internal_name (*mp_output_filename*, *xstrdup* ("outputfilename"));
  set_internal_name (*mp_output_format*, *xstrdup* ("outputformat"));
  set_internal_name (*mp_output_format_options*, *xstrdup* ("outputformatoptions"));
  set_internal_name (*mp_job_name*, *xstrdup* ("jobname"));
  set_internal_name (*mp_number_system*, *xstrdup* ("numbersystem"));
  set_internal_name (*mp_number_precision*, *xstrdup* ("numberprecision"));
  set_internal_name (*mp_hppp*, *xstrdup* ("hppp"));
  set_internal_name (*mp_vppp*, *xstrdup* ("vppp"));

204.   The following procedure, which is called just before METAPOST initializes its input and output, establishes the initial values of the date and time.

Note that the values are *scaled* integers. Hence METAPOST can no longer be used after the year 32767.

static void *mp_fix_date_and_time*(**MP** *mp*)
{
    time_t *aclock* = *time*((**time_t** *) 0);
    struct *tm* *∗tmptr* = *localtime*(&*aclock*);

    set_internal_from_number(*mp_time*, *unity_t*);
    number_multiply_int(*internal_value*(*mp_time*), (*tmptr*→*tm_hour* ∗ 60 + *tmptr*→*tm_min*));
    set_internal_from_number(*mp_hour*, *unity_t*);
    number_multiply_int(*internal_value*(*mp_hour*), (*tmptr*→*tm_hour*));
    set_internal_from_number(*mp_minute*, *unity_t*);
    number_multiply_int(*internal_value*(*mp_minute*), (*tmptr*→*tm_min*));
    set_internal_from_number(*mp_day*, *unity_t*);
    number_multiply_int(*internal_value*(*mp_day*), (*tmptr*→*tm_mday*));
    set_internal_from_number(*mp_month*, *unity_t*);
    number_multiply_int(*internal_value*(*mp_month*), (*tmptr*→*tm_mon* + 1));
    set_internal_from_number(*mp_year*, *unity_t*);
    number_multiply_int(*internal_value*(*mp_year*), (*tmptr*→*tm_year* + 1900));
}

205.   ⟨ Declarations 8 ⟩ +≡
    static void *mp_fix_date_and_time*(**MP** *mp*);

206.   METAPOST is occasionally supposed to print diagnostic information that goes only into the transcript file, unless *mp_tracing_online* is positive. Now that we have defined *mp_tracing_online* we can define two routines that adjust the destination of print commands:

⟨ Declarations 8 ⟩ +≡
    static void *mp_begin_diagnostic*(**MP** *mp*);
    static void *mp_end_diagnostic*(**MP** *mp*, **boolean** *blank_line*);
    static void *mp_print_diagnostic*(**MP** *mp*, **const char** *∗s*, **const char** *∗t*, **boolean** *nuline*);

207.   ⟨ Basic printing procedures 85 ⟩ +≡
    void *mp_begin_diagnostic*(**MP** *mp*)
    {   /∗ prepare to do some tracing ∗/
        mp→*old_setting* = *mp*→*selector*;
        if (*number_nonpositive*(*internal_value*(*mp_tracing_online*)) ∧ (*mp*→*selector* ≡ *term_and_log*)) {
            decr(*mp*→*selector*);
            if (*mp*→*history* ≡ *mp_spotless*) *mp*→*history* = *mp_warning_issued*;
        }
    }

    void *mp_end_diagnostic*(**MP** *mp*, **boolean** *blank_line*)
    {   /∗ restore proper conditions after tracing ∗/
        mp_print_nl(*mp*, "");
        if (*blank_line*) *mp_print_ln*(*mp*);
        mp→*selector* = *mp*→*old_setting*;
    }

208.
⟨ Global variables 14 ⟩ +≡
    unsigned int *old_setting*;

209.   We will occasionally use *begin_diagnostic* in connection with line-number printing, as follows. (The parameter *s* is typically `"Path"` or `"Cycle␣spec"`, etc.)

⟨ Basic printing procedures 85 ⟩ +≡
  void *mp_print_diagnostic*(**MP** *mp*, **const char** ∗*s*, **const char** ∗*t*, **boolean** *nuline*)
  {
    mp_begin_diagnostic(*mp*);
    if (*nuline*) *mp_print_nl*(*mp*, *s*);
    else *mp_print*(*mp*, *s*);
    mp_print(*mp*, "␣at␣line␣");
    mp_print_int(*mp*, *mp_true_line*(*mp*));
    mp_print(*mp*, *t*);
    mp_print_char(*mp*, *xord*(':'));
  }

210.   The 256 **ASCII_code** characters are grouped into classes by means of the *char_class* table. Individual class numbers have no semantic or syntactic significance, except in a few instances defined here. There's also *max_class*, which can be used as a basis for additional class numbers in nonstandard extensions of METAPOST.

#**define** *digit_class* 0   /∗ the class number of `0123456789` ∗/
#**define** *period_class* 1   /∗ the class number of '.' ∗/
#**define** *space_class* 2   /∗ the class number of spaces and nonstandard characters ∗/
#**define** *percent_class* 3   /∗ the class number of '`%`' ∗/
#**define** *string_class* 4   /∗ the class number of '"' ∗/
#**define** *right_paren_class* 8   /∗ the class number of ')' ∗/
#**define** *isolated_classes* 5: **case** 6: **case** 7: **case** 8
            /∗ characters that make length-one tokens only ∗/
#**define** *letter_class* 9   /∗ letters and the underline character ∗/
#**define** *mp_left_bracket_class* 17   /∗ '[' ∗/
#**define** *mp_right_bracket_class* 18   /∗ ']' ∗/
#**define** *invalid_class* 20   /∗ bad character in the input ∗/
#**define** *max_class* 20   /∗ the largest class number ∗/

⟨ Global variables 14 ⟩ +≡
#**define** *digit_class* 0   /∗ the class number of `0123456789` ∗/
  int *char_class*[256];   /∗ the class numbers ∗/

211.   If changes are made to accommodate non-ASCII character sets, they should follow the guidelines in Appendix C of *The METAFONT book*.

⟨ Set initial values of key variables 38 ⟩ +≡

  for $(k = \text{'0'}; \; k \leq \text{'9'}; \; k{+}{+})$ $mp{\to}char\_class[k] = digit\_class$;

  $mp{\to}char\_class[\text{'.'}] = period\_class$;

  $mp{\to}char\_class[\text{'␣'}] = space\_class$;

  $mp{\to}char\_class[\text{'%'}] = percent\_class$;

  $mp{\to}char\_class[\text{'"'}] = string\_class$;

  $mp{\to}char\_class[\text{','}] = 5$;

  $mp{\to}char\_class[\text{';'}] = 6$;

  $mp{\to}char\_class[\text{'('}] = 7$;

  $mp{\to}char\_class[\text{')'}] = right\_paren\_class$;

  for $(k = \text{'A'}; \; k \leq \text{'Z'}; \; k{+}{+})$ $mp{\to}char\_class[k] = letter\_class$;

  for $(k = \text{'a'}; \; k \leq \text{'z'}; \; k{+}{+})$ $mp{\to}char\_class[k] = letter\_class$;

  $mp{\to}char\_class[\text{'\_'}] = letter\_class$;

  $mp{\to}char\_class[\text{'<'}] = 10$;

  $mp{\to}char\_class[\text{'='}] = 10$;

  $mp{\to}char\_class[\text{'>'}] = 10$;

  $mp{\to}char\_class[\text{':'}] = 10$;

  $mp{\to}char\_class[\text{'|'}] = 10$;

  $mp{\to}char\_class[\text{'`'}] = 11$;

  $mp{\to}char\_class[\text{'\''}] = 11$;

  $mp{\to}char\_class[\text{'+'}] = 12$;

  $mp{\to}char\_class[\text{'-'}] = 12$;

  $mp{\to}char\_class[\text{'/'}] = 13$;

  $mp{\to}char\_class[\text{'*'}] = 13$;

  $mp{\to}char\_class[\text{'\\'}] = 13$;

  $mp{\to}char\_class[\text{'!'}] = 14$;

  $mp{\to}char\_class[\text{'?'}] = 14$;

  $mp{\to}char\_class[\text{'#'}] = 15$;

  $mp{\to}char\_class[\text{'&'}] = 15$;

  $mp{\to}char\_class[\text{'@'}] = 15$;

  $mp{\to}char\_class[\text{'$'}] = 15$;

  $mp{\to}char\_class[\text{'\^'}] = 16$;

  $mp{\to}char\_class[\text{'~'}] = 16$;

  $mp{\to}char\_class[\text{'['}] = mp\_left\_bracket\_class$;

  $mp{\to}char\_class[\text{']'}] = mp\_right\_bracket\_class$;

  $mp{\to}char\_class[\text{'\{'}] = 19$;

  $mp{\to}char\_class[\text{'\}'}] = 19$;

  for $(k = 0; \; k < \text{'␣'}; \; k{+}{+})$ $mp{\to}char\_class[k] = invalid\_class$;

  $mp{\to}char\_class[\text{'\t'}] = space\_class$;

  $mp{\to}char\_class[\text{'\f'}] = space\_class$;

  for $(k = 127; \; k \leq 255; \; k{+}{+})$ $mp{\to}char\_class[k] = invalid\_class$;

212.   The hash table.
 Symbolic tokens are stored in and retrieved from an AVL tree. This is not as fast as an actual hash table, but it is easily extensible.
 A symbolic token contains a pointer to the **mp_string** that contains the string representation of the symbol, a **halfword** that holds the current command value of the token, and an **mp_value** for the associated equivalent.

#define $set\_text(A)$ **do**
            {
               FUNCTION_TRACE3("set_text(%p,␣%p)\n", $(A), (B)$);
               (A)‑$text = (B)$;
            }
            while (0)
#define $set\_eq\_type(A, B)$ **do**
            {
               FUNCTION_TRACE3("set_eq_type(%p,␣%d)\n", $(A), (B)$);
               (A)‑$type = (B)$;
            }
            while (0)
#define $set\_equiv(A, B)$ **do**
            {
               FUNCTION_TRACE3("set_equiv(%p,␣%d)\n", $(A), (B)$);
               (A)‑$v.data.node = \Lambda$;
               (A)‑$v.data.indep.serial = (B)$;
            }
            while (0)
#define $set\_equiv\_node(A, B)$ **do**
            {
               FUNCTION_TRACE3("set_equiv_node(%p,␣%p)\n", $(A), (B)$);
               (A)‑$v.data.node = (B)$;
               (A)‑$v.data.indep.serial = 0$;
            }
            while (0)
#define $set\_equiv\_sym(A, B)$ **do**
            {
               FUNCTION_TRACE3("set_equiv_sym(%p,␣%p)\n", $(A), (B)$);
               (A)‑$v.data.node = ($**mp_node**$)(B)$;
               (A)‑$v.data.indep.serial = 0$;
            }
            while (0)

213.

#**if** DEBUG
#**define** $text(A)do\_get\_text$ $(mp,(A))$
#**define** $eq\_type(A)do\_get\_eq\_type$ $(mp,(A))$
#**define** $equiv(A)do\_get\_equiv$ $(mp,(A))$
#**define** $equiv\_node(A)do\_get\_equiv\_node$ $(mp,(A))$
#**define** $equiv\_sym(A)do\_get\_equiv\_sym$ $(mp,(A))$
  static mp_string do_get_text(**MP** mp, **mp_sym** A)
  {
    FUNCTION_TRACE3("%d␣=␣do_get_text(%p)\n", $A{\rightarrow}text$, A);
    return $A{\rightarrow}text$;
  }
  static halfword $do\_get\_eq\_type$(**MP** mp, **mp_sym** A)
  {
    FUNCTION_TRACE3("%d␣=␣do_get_eq_type(%p)\n", $A{\rightarrow}type$, A);
    return $A{\rightarrow}type$;
  }
  static halfword do_get_equiv(**MP** mp, **mp_sym** A)
  {
    FUNCTION_TRACE3("%d␣=␣do_get_equiv(%p)\n", $A{\rightarrow}v.data.indep.serial$, A);
    return $A{\rightarrow}v.data.indep.serial$;
  }
  static mp_node $do\_get\_equiv\_node$(**MP** mp, **mp_sym** A)
  {
    FUNCTION_TRACE3("%p␣=␣do_get_equiv_node(%p)\n", $A{\rightarrow}v.data.node$, A);
    return $A{\rightarrow}v.data.node$;
  }
  static mp_sym $do\_get\_equiv\_sym$(**MP** mp, **mp_sym** A)
  {
    FUNCTION_TRACE3("%p␣=␣do_get_equiv_sym(%p)\n", $A{\rightarrow}v.data.node$, A);
    return (**mp_sym**) $A{\rightarrow}v.data.node$;
  }
#**else**
#**define** $text(A)(A){\rightarrow}text$
#**define** $eq\_type(A)(A){\rightarrow}type$
#**define** $equiv(A)(A){\rightarrow}v.data.indep.serial$
#**define** $equiv\_node(A)(A){\rightarrow}v.data.node$
#**define** $equiv\_sym(A)(\textbf{mp\_sym})(A){\rightarrow}v.data.node$
#**endif**

214.   ⟨ Declarations 8 ⟩ +≡
#**if** DEBUG
  static mp_string do_get_text(**MP** mp, **mp_sym** A);
  static halfword $do\_get\_eq\_type$(**MP** mp, **mp_sym** A);
  static halfword do_get_equiv(**MP** mp, **mp_sym** A);
  static mp_node $do\_get\_equiv\_node$(**MP** mp, **mp_sym** A);
  static mp_sym $do\_get\_equiv\_sym$(**MP** mp, **mp_sym** A);
#**endif**

215.   ⟨Types in the outer block 33⟩ +≡
   typedef struct mp_symbol_entry {
      halfword *type*;
      mp_value *v*;
      mp_string *text*;
      void *∗parent*;
   } **mp_symbol_entry**;

216.   ⟨Global variables 14⟩ +≡
   integer *st_count*;   /∗ total number of known identifiers ∗/

   avl_tree symbols;   /∗ avl tree of symbolic tokens ∗/
   avl_tree frozen_symbols;   /∗ avl tree of frozen symbolic tokens ∗/

   mp_sym *frozen_bad_vardef*;
   mp_sym *frozen_colon*;
   mp_sym *frozen_end_def*;
   mp_sym *frozen_end_for*;
   mp_sym *frozen_end_group*;
   mp_sym *frozen_etex*;
   mp_sym *frozen_fi*;
   mp_sym *frozen_inaccessible*;
   mp_sym *frozen_left_bracket*;
   mp_sym *frozen_mpx_break*;
   mp_sym *frozen_repeat_loop*;
   mp_sym *frozen_right_delimiter*;
   mp_sym *frozen_semicolon*;
   mp_sym *frozen_slash*;
   mp_sym *frozen_undefined*;
   mp_sym *frozen_dump*;

217.   Here are the functions needed for the avl construction.

⟨Declarations 8⟩ +≡
   static int *comp_symbols_entry*(**void** *∗p*, **const void** *∗pa*, **const void** *∗pb*);
   static void *∗copy_symbols_entry*(**const void** *∗p*);
   static void *∗delete_symbols_entry*(**void** *∗p*);

218.   The avl comparison function is a straightword version of *strcmp*, except that checks for the string lengths first.
   static int *comp_symbols_entry*(**void** *∗p*, **const void** *∗pa*, **const void** *∗pb*)
   {
      const mp_symbol_entry *∗a* = (**const mp_symbol_entry** *∗*) *pa*;
      const mp_symbol_entry *∗b* = (**const mp_symbol_entry** *∗*) *pb*;

      (**void**) *p*;
      if (*a→text→len* ≠ *b→text→len*) {
         return (*a→text→len* > *b→text→len* ? 1 : −1);
      }
      return *strncmp*((**const char** *∗*) *a→text→str*, (**const char** *∗*) *b→text→str*, *a→text→len*);
   }

219.   Copying a symbol happens when an item is inserted into an AVL tree. The *text* and **mp_number** needs to be deep copied, every thing else can be reassigned.

```
static void *copy_symbols_entry(const void *p)
{
  MP mp;
  mp_sym ff;
  const mp_symbol_entry *fp;

  fp = (const mp_symbol_entry *) p;
  mp = (MP) fp→parent;
  ff = malloc(sizeof(mp_symbol_entry));
  if (ff ≡ Λ) return Λ;
  ff→text = copy_strings_entry(fp→text);
  if (ff→text ≡ Λ) return Λ;
  ff→v = fp→v;
  ff→type = fp→type;
  ff→parent = mp;
  new_number(ff→v.data.n);
  number_clone(ff→v.data.n, fp→v.data.n);
  return ff;
}
```

220.   In the current implementation, symbols are not freed until the end of the run.

```
static void *delete_symbols_entry(void *p)
{
  MP mp;
  mp_sym ff = (mp_sym) p;

  mp = (MP) ff→parent;
  free_number(ff→v.data.n);
  mp_xfree(ff→text→str);
  mp_xfree(ff→text);
  mp_xfree(ff);
  return Λ;
}
```

221.   ⟨ Allocate or initialize variables 28 ⟩ +≡
  mp→symbols = avl_create(comp_symbols_entry, copy_symbols_entry, delete_symbols_entry, malloc, free, Λ);
  mp→frozen_symbols = avl_create(comp_symbols_entry, copy_symbols_entry, delete_symbols_entry, malloc,
        free, Λ);

222.   ⟨ Dealloc variables 27 ⟩ +≡
  if (mp→symbols ≠ Λ) avl_destroy(mp→symbols);
  if (mp→frozen_symbols ≠ Λ) avl_destroy(mp→frozen_symbols);

223.   Actually creating symbols is done by *id_lookup*, but in order to do so it needs a way to create a new, empty symbol structure.

⟨ Declarations 8 ⟩ +≡
  static mp_sym new_symbols_entry(MP mp, unsigned char *nam, size_t len);

224.   **static mp_sym** *new_symbols_entry*(**MP** *mp*, **unsigned char** *∗nam*, **size_t** *len*)
  {
    mp_sym *ff*;
    ff = *mp_xmalloc*(*mp*, 1, **sizeof**(**mp_symbol_entry**));
    memset(*ff*, 0, **sizeof**(**mp_symbol_entry**));
    ff→*parent* = *mp*;
    ff→*text* = *mp_xmalloc*(*mp*, 1, **sizeof**(**mp_lstring**));
    ff→*text*→*str* = *nam*;
    ff→*text*→*len* = *len*;
    ff→*type* = *mp_tag_token*;
    ff→*v.type* = *mp_known*;
    new_number(*ff*→*v.data.n*);
    FUNCTION_TRACE4("%p␣=␣new_symbols_entry(\"%s\",%d)\n", *ff*, *nam*, (**int**) *len*);
    return *ff*;
  }

225.   There is one global variable so that *id_lookup* does not always have to create a new entry just for
testing. This is not freed because it creates a double-free thanks to the Λ init.

⟨ Global variables 14 ⟩ +≡
  mp_sym *id_lookup_test*;

226.   ⟨ Initialize table entries 182 ⟩ +≡
  mp→*id_lookup_test* = *new_symbols_entry*(*mp*, Λ, 0);

227.   Certain symbols are "frozen" and not redefinable, since they are used in error recovery.

⟨ Initialize table entries 182 ⟩ +≡
  mp→*st_count* = 0;
  mp→*frozen_bad_vardef* = *mp_frozen_primitive*(*mp*, "a␣bad␣variable", *mp_tag_token*, 0);
  mp→*frozen_right_delimiter* = *mp_frozen_primitive*(*mp*, ")", *mp_right_delimiter*, 0);
  mp→*frozen_inaccessible* = *mp_frozen_primitive*(*mp*, "␣INACCESSIBLE", *mp_tag_token*, 0);
  mp→*frozen_undefined* = *mp_frozen_primitive*(*mp*, "␣UNDEFINED", *mp_tag_token*, 0);

228.   Here is the subroutine that searches the avl tree for an identifier that matches a given string of
length l appearing in $buffer[j \mathrel{..} (j+l-1)]$. If the identifier is not found, it is inserted if $insert\_new$ is $true$,
and the corresponding symbol will be returned.

There are two variations on the lookup function: one for the normal symbol table, and one for the table
of error recovery symbols.

#define $mp\_id\_lookup(A, B, C, D)$ $mp\_do\_id\_lookup((A), mp \rightarrow symbols, (B), (C), (D))$

  static mp_sym $mp\_do\_id\_lookup(\textbf{MP}\ mp, \textbf{avl\_tree}\ symbols, \textbf{char}\ *j, \textbf{size\_t}\ l, \textbf{boolean}\ insert\_new)$
  $\{$ /∗ search an avl tree ∗/
    mp_sym str;

    $mp \rightarrow id\_lookup\_test \rightarrow text \rightarrow str = (\textbf{unsigned char}\ *)\ j$;
    $mp \rightarrow id\_lookup\_test \rightarrow text \rightarrow len = l$;
    $str = (\textbf{mp\_sym})\ avl\_find(mp \rightarrow id\_lookup\_test, symbols)$;
    if $(str \equiv \Lambda \wedge insert\_new)\ \{$
      unsigned char $*nam = (\textbf{unsigned char}\ *)\ mp\_xstrldup(mp, j, l)$;
      mp_sym $s = new\_symbols\_entry(mp, nam, l)$;

      $mp \rightarrow st\_count +\!+$;
      $assert(avl\_ins(s, symbols, avl\_false) > 0)$;
      $str = (\textbf{mp\_sym})\ avl\_find(s, symbols)$;
      $delete\_symbols\_entry(s)$;
    $\}$
    return str;
  $\}$

  static mp_sym $mp\_frozen\_id\_lookup(\textbf{MP}\ mp, \textbf{char}\ *j, \textbf{size\_t}\ l, \textbf{boolean}\ insert\_new)$
  $\{$ /∗ search the error recovery symbol table ∗/
    return $mp\_do\_id\_lookup(mp, mp \rightarrow frozen\_symbols, j, l, insert\_new)$;
  $\}$

229.   We need to put METAPOST's "primitive" symbolic tokens into the hash table, together with their
command code (which will be the eq_type) and an operand (which will be the $equiv$). The $primitive$ procedure
does this, in a way that no METAPOST user can. The global value cur_sym contains the new $eqtb$ pointer
after $primitive$ has acted.

  static void $mp\_primitive(\textbf{MP}\ mp, \textbf{const char}\ *ss, \textbf{halfword}\ c, \textbf{halfword}\ o)$
  $\{$
    char $*s = mp\_xstrdup(mp, ss)$;
    $set\_cur\_sym(mp\_id\_lookup(mp, s, strlen(s), true))$;
    $mp\_xfree(s)$;
    $set\_eq\_type(cur\_sym(), c)$;
    $set\_equiv(cur\_sym(), o)$;
  $\}$

230.   Some other symbolic tokens only exist for error recovery.

  static mp_sym $mp\_frozen\_primitive(\textbf{MP}\ mp, \textbf{const char}\ *ss, \textbf{halfword}\ c, \textbf{halfword}\ o)$
  $\{$
    char $*s = mp\_xstrdup(mp, ss)$;
    mp_sym $str = mp\_frozen\_id\_lookup(mp, s, strlen(ss), true)$;

    $mp\_xfree(s)$;
    $str \rightarrow type = c$;
    $str \rightarrow v.data.indep.serial = o$;
    return str;
  $\}$

231.   This routine returns *true* if the argument is an un-redefinable symbol because it is one of the error recovery tokens (as explained elsewhere, *frozen_inaccessible* actuall is redefinable).

```
static boolean mp_is_frozen(MP mp, mp_sym sym)
{
    mp_sym temp = mp_frozen_id_lookup(mp, (char *) sym→text→str, sym→text→len, false);
    if (temp ≡ mp→frozen_inaccessible) return false;
    return (temp ≡ sym);
}
```

232.   Many of METAPOST's primitives need no *equiv*, since they are identifiable by their *eq_type* alone. These primitives are loaded into the hash table as follows:

⟨ Put each of METAPOST's primitives into the hash table 200 ⟩ +≡
$mp\_primitive(mp, "..", mp\_path\_join, 0);$
;
$mp\_primitive(mp, "[", mp\_left\_bracket, 0);$
$mp\rightarrow frozen\_left\_bracket = mp\_frozen\_primitive(mp, "[", mp\_left\_bracket, 0);$
;
$mp\_primitive(mp, "]", mp\_right\_bracket, 0);$
;
$mp\_primitive(mp, "\}", mp\_right\_brace, 0);$
;
$mp\_primitive(mp, "\{", mp\_left\_brace, 0);$
;
$mp\_primitive(mp, ":", mp\_colon, 0);$
$mp\rightarrow frozen\_colon = mp\_frozen\_primitive(mp, ":", mp\_colon, 0);$
;
$mp\_primitive(mp, "::", mp\_double\_colon, 0);$
;
$mp\_primitive(mp, "||:", mp\_bchar\_label, 0);$
;
$mp\_primitive(mp, ":=", mp\_assignment, 0);$
;
$mp\_primitive(mp, ",", mp\_comma, 0);$
;
$mp\_primitive(mp, ";", mp\_semicolon, 0);$
$mp\rightarrow frozen\_semicolon = mp\_frozen\_primitive(mp, ";", mp\_semicolon, 0);$
;
$mp\_primitive(mp, "\backslash\backslash", mp\_relax, 0);$
;
$mp\_primitive(mp, "addto", mp\_add\_to\_command, 0);$
;
$mp\_primitive(mp, "atleast", mp\_at\_least, 0);$
;
$mp\_primitive(mp, "begingroup", mp\_begin\_group, 0);$
$mp\rightarrow bg\_loc = cur\_sym();$
;
$mp\_primitive(mp, "controls", mp\_controls, 0);$
;
$mp\_primitive(mp, "curl", mp\_curl\_command, 0);$
;
$mp\_primitive(mp, "delimiters", mp\_delimiters, 0);$
;
$mp\_primitive(mp, "endgroup", mp\_end\_group, 0);$
$mp\rightarrow eg\_loc = cur\_sym();$
$mp\rightarrow frozen\_end\_group = mp\_frozen\_primitive(mp, "endgroup", mp\_end\_group, 0);$
;
$mp\_primitive(mp, "everyjob", mp\_every\_job\_command, 0);$
;
$mp\_primitive(mp, "exitif", mp\_exit\_test, 0);$
;
$mp\_primitive(mp, "expandafter", mp\_expand\_after, 0);$

;
$mp\_primitive(mp, \texttt{"interim"}, mp\_interim\_command, 0);$
;
$mp\_primitive(mp, \texttt{"let"}, mp\_let\_command, 0);$
;
$mp\_primitive(mp, \texttt{"newinternal"}, mp\_new\_internal, 0);$
;
$mp\_primitive(mp, \texttt{"of"}, mp\_of\_token, 0);$
;
$mp\_primitive(mp, \texttt{"randomseed"}, mp\_random\_seed, 0);$
;
$mp\_primitive(mp, \texttt{"save"}, mp\_save\_command, 0);$
;
$mp\_primitive(mp, \texttt{"scantokens"}, mp\_scan\_tokens, 0);$
;
$mp\_primitive(mp, \texttt{"shipout"}, mp\_ship\_out\_command, 0);$
;
$mp\_primitive(mp, \texttt{"skipto"}, mp\_skip\_to, 0);$
;
$mp\_primitive(mp, \texttt{"special"}, mp\_special\_command, 0);$
;
$mp\_primitive(mp, \texttt{"fontmapfile"}, mp\_special\_command, 1);$
;
$mp\_primitive(mp, \texttt{"fontmapline"}, mp\_special\_command, 2);$
;
$mp\_primitive(mp, \texttt{"step"}, mp\_step\_token, 0);$
;
$mp\_primitive(mp, \texttt{"str"}, mp\_str\_op, 0);$
;
$mp\_primitive(mp, \texttt{"tension"}, mp\_tension, 0);$
;
$mp\_primitive(mp, \texttt{"to"}, mp\_to\_token, 0);$
;
$mp\_primitive(mp, \texttt{"until"}, mp\_until\_token, 0);$
;
$mp\_primitive(mp, \texttt{"within"}, mp\_within\_token, 0);$
;
$mp\_primitive(mp, \texttt{"write"}, mp\_write\_command, 0);$

233.   Each primitive has a corresponding inverse, so that it is possible to display the cryptic numeric contents of *eqtb* in symbolic form. Every call of *primitive* in this program is therefore accompanied by some straightforward code that forms part of the *print_cmd_mod* routine explained below.

⟨ Cases of *print_cmd_mod* for symbolic printing of primitives 233 ⟩ ≡
case *mp_add_to_command*: *mp_print*(*mp*, "addto");
  break;
case *mp_assignment*: *mp_print*(*mp*, ":=");
  break;
case *mp_at_least*: *mp_print*(*mp*, "atleast");
  break;
case *mp_bchar_label*: *mp_print*(*mp*, "||:");
  break;
case *mp_begin_group*: *mp_print*(*mp*, "begingroup");
  break;
case *mp_colon*: *mp_print*(*mp*, ":");
  break;
case *mp_comma*: *mp_print*(*mp*, ",");
  break;
case *mp_controls*: *mp_print*(*mp*, "controls");
  break;
case *mp_curl_command*: *mp_print*(*mp*, "curl");
  break;
case *mp_delimiters*: *mp_print*(*mp*, "delimiters");
  break;
case *mp_double_colon*: *mp_print*(*mp*, "::");
  break;
case *mp_end_group*: *mp_print*(*mp*, "endgroup");
  break;
case *mp_every_job_command*: *mp_print*(*mp*, "everyjob");
  break;
case *mp_exit_test*: *mp_print*(*mp*, "exitif");
  break;
case *mp_expand_after*: *mp_print*(*mp*, "expandafter");
  break;
case *mp_interim_command*: *mp_print*(*mp*, "interim");
  break;
case *mp_left_brace*: *mp_print*(*mp*, "{");
  break;
case *mp_left_bracket*: *mp_print*(*mp*, "[");
  break;
case *mp_let_command*: *mp_print*(*mp*, "let");
  break;
case *mp_new_internal*: *mp_print*(*mp*, "newinternal");
  break;
case *mp_of_token*: *mp_print*(*mp*, "of");
  break;
case *mp_path_join*: *mp_print*(*mp*, "..");
  break;
case *mp_random_seed*: *mp_print*(*mp*, "randomseed");
  break;
case *mp_relax*: *mp_print_char*(*mp*, *xord*('\\'));
  break;

case mp_right_brace: $mp\_print\_char(mp, xord(\texttt{'}\texttt{\}'}))$;
  break;
case $mp\_right\_bracket$: $mp\_print\_char(mp, xord(\texttt{']'}))$;
  break;
case $mp\_save\_command$: $mp\_print(mp, \texttt{"save"})$;
  break;
case mp_scan_tokens: $mp\_print(mp, \texttt{"scantokens"})$;
  break;
case $mp\_semicolon$: $mp\_print\_char(mp, xord(\texttt{';'}))$;
  break;
case $mp\_ship\_out\_command$: $mp\_print(mp, \texttt{"shipout"})$;
  break;
case mp_skip_to: $mp\_print(mp, \texttt{"skipto"})$;
  break;
case $mp\_special\_command$:
  if $(m \equiv 2)$ $mp\_print(mp, \texttt{"fontmapline"})$;
  else if $(m \equiv 1)$ $mp\_print(mp, \texttt{"fontmapfile"})$;
  else $mp\_print(mp, \texttt{"special"})$;
  break;
case mp_step_token: $mp\_print(mp, \texttt{"step"})$;
  break;
case mp_str_op: $mp\_print(mp, \texttt{"str"})$;
  break;
case $mp\_tension$: $mp\_print(mp, \texttt{"tension"})$;
  break;
case mp_to_token: $mp\_print(mp, \texttt{"to"})$;
  break;
case mp_until_token: $mp\_print(mp, \texttt{"until"})$;
  break;
case mp_within_token: $mp\_print(mp, \texttt{"within"})$;
  break;
case $mp\_write\_command$: $mp\_print(mp, \texttt{"write"})$;
  break;

See also sections 736, 746, 754, 760, 772, 810, 956, 1047, 1072, 1079, 1082, 1100, 1106, 1123, 1129, 1144, 1176, and 1186.

This code is used in section 671.

234.   We will deal with the other primitives later, at some point in the program where their *eq_type* and *equiv* values are more meaningful. For example, the primitives for macro definitions will be loaded when we consider the routines that define macros. It is easy to find where each particular primitive was treated by looking in the index at the end; for example, the section where $\texttt{"def"}$ entered *eqtb* is listed under '**def** primitive'.

235.   Token lists.

A METAPOST token is either symbolic or numeric or a string, or it denotes a macro parameter or capsule or an internal; so there are six corresponding ways to encode it internally:

(1) A symbolic token for symbol p is represented by the pointer p, in the *sym_sym* field of a symbolic node in *mem*. The *type* field is *symbol_node*; and it has a *name_type* to differentiate various subtypes of symbolic tokens, which is usually *normal_sym*, but *macro_sym* for macro names.

(2) A numeric token whose *scaled* value is v is represented in a non-symbolic node of *mem*; the *type* field is *known*, the *name_type* field is *token*, and the *value* field holds v.

(3) A string token is also represented in a non-symbolic node; the *type* field is *mp_string_type*, the *name_type* field is *token*, and the *value* field holds the corresponding **mp_string**.

(4) Capsules have *name_type* = *capsule*, and their *type* and *value* fields represent arbitrary values, with *type* different from *symbol_node* (in ways to be explained later).

(5) Macro parameters appear in *sym_info* fields of symbolic nodes. The *type* field is *symbol_node*; the kth parameter is represented by k in *sym_info*; and *expr_sym* in *name_type*, if it is of type **expr**, or *suffix_sym* if it is of type **suffix**, or by *text_sym* if it is of type **text**.

(6) The kth internal is also represented by k in *sym_info*; the *type* field is *symbol_node* as for the other symbolic tokens; and *internal_sym* is its *name_type*;

Actual values of the parameters and internals are kept in a separate stack, as we will see later.

Note that the '*type*' field of a node has nothing to do with "type" in a printer's sense. It's curious that the same word is used in such different ways.

#define *token_node_size* **sizeof**(**mp_node_data**)   /∗ the number of words in a large token node ∗/
#define *set_value_sym*(*A, B*)   *do_set_value_sym*(*mp*, (*mp_token_node*)(*A*), (*B*))
#define *set_value_number*(*A, B*)   *do_set_value_number*(*mp*, (*mp_token_node*)(*A*), (*B*))
#define *set_value_node*(*A, B*)   *do_set_value_node*(*mp*, (*mp_token_node*)(*A*), (*B*))
#define *set_value_str*(*A, B*)   *do_set_value_str*(*mp*, (*mp_token_node*)(*A*), (*B*))
#define *set_value_knot*(*A, B*)   *do_set_value_knot*(*mp*, (*mp_token_node*)*A*, (*B*))
#define *value_sym_NEW*(*A*)   (**mp_sym**) *mp_link*(*A*)
#define *set_value_sym_NEW*(*A, B*)   *set_mp_link*(*A*, (**mp_node**) *B*)

⟨MPlib internal header stuff 6⟩ +≡
  typedef struct mp_node_data ∗**mp_token_node**;

236.

```
#if DEBUG
#define value_sym(A)do_get_value_sym    (mp, (mp_token_node)(A))
    /* # define value_number(A)do_get_value_number(mp, (mp_token_node)(A)) */
#define value_number(A)((mp_token_node)(A))⃗data.n
#define value_node(A)do_get_value_node    (mp, (mp_token_node)(A))
#define value_str(A)do_get_value_str    (mp, (mp_token_node)(A))
#define value_knot(A)do_get_value_knot    (mp, (mp_token_node)(A))
#else
#define value_sym(A)((mp_token_node)(A))⃗data.sym
#define value_number(A)((mp_token_node)(A))⃗data.n
#define value_node(A)((mp_token_node)(A))⃗data.node
#define value_str(A)((mp_token_node)(A))⃗data.str
#define value_knot(A)((mp_token_node)(A))⃗data.p
#endif
  static void do_set_value_sym(MP mp, mp_token_node A, mp_sym B)
  {
    FUNCTION_TRACE3("set_value_sym(%p,%p)\n", (A), (B));
    A⃗data.sym = (B);
  }

  static void do_set_value_number(MP mp, mp_token_node A, mp_number B)
  {
    FUNCTION_TRACE3("set_value(%p,%s)\n", (A), number_tostring(B));
    A⃗data.p = Λ;
    A⃗data.str = Λ;
    A⃗data.node = Λ;
    number_clone(A⃗data.n, B);
  }

  static void do_set_value_str(MP mp, mp_token_node A, mp_string B)
  {
    FUNCTION_TRACE3("set_value_str(%p,%p)\n", (A), (B));
    assert(A⃗type ≠ mp_structured);
    A⃗data.p = Λ;
    A⃗data.str = (B);
    add_str_ref((B));
    A⃗data.node = Λ;
    number_clone(A⃗data.n, zero_t);
  }

  static void do_set_value_node(MP mp, mp_token_node A, mp_node B)
  {     /* store the value in a large token node */
    FUNCTION_TRACE3("set_value_node(%p,%p)\n", A, B);
    assert(A⃗type ≠ mp_structured);
    A⃗data.p = Λ;
    A⃗data.str = Λ;
    A⃗data.node = B;
    number_clone(A⃗data.n, zero_t);
  }

  static void do_set_value_knot(MP mp, mp_token_node A, mp_knot B)
  {
    FUNCTION_TRACE3("set_value_knot(%p,%p)\n", (A), (B));
    assert(A⃗type ≠ mp_structured);
```

```
    A→data.p = (B);
    A→data.str = Λ;
    A→data.node = Λ;
    number_clone(A→data.n, zero_t);
  }
```

237.
#if DEBUG
  static mp_sym *do_get_value_sym*(**MP** *mp*, **mp_token_node** *A*)
  {   /∗ *A→type* can be structured in this case ∗/
    FUNCTION_TRACE3("%p␣=␣get_value_sym(%p)\n", *A→data.sym*, *A*);
    return *A→data.sym*;
  }
  static mp_node *do_get_value_node*(**MP** *mp*, **mp_token_node** *A*)
  {
    assert(*A→type* ≠ *mp_structured*);
    FUNCTION_TRACE3("%p␣=␣get_value_node(%p)\n", *A→data.node*, *A*);
    return *A→data.node*;
  }
  static mp_string *do_get_value_str*(**MP** *mp*, **mp_token_node** *A*)
  {
    assert(*A→type* ≠ *mp_structured*);
    FUNCTION_TRACE3("%p␣=␣get_value_str(%p)\n", *A→data.str*, *A*);
    return *A→data.str*;
  }
  static *mp_knot do_get_value_knot*(**MP** *mp*, **mp_token_node** *A*)
  {
    assert(*A→type* ≠ *mp_structured*);
    FUNCTION_TRACE3("%p␣=␣get_value_knot(%p)\n", *A→data.p*, *A*);
    return *A→data.p*;
  }
  static mp_number *do_get_value_number*(**MP** *mp*, **mp_token_node** *A*)
  {
    assert(*A→type* ≠ *mp_structured*);
    FUNCTION_TRACE3("%d␣=␣get_value_number(%p)\n", *A→data.n.type*, *A*);
    return *A→data.n*;
  }
#endif

238.   ⟨Declarations 8⟩ +≡
#if DEBUG
  static mp_number *do_get_value_number*(**MP** *mp*, **mp_token_node** *A*);
  static mp_sym *do_get_value_sym*(**MP** *mp*, **mp_token_node** *A*);
  static mp_node *do_get_value_node*(**MP** *mp*, **mp_token_node** *A*);
  static mp_string *do_get_value_str*(**MP** *mp*, **mp_token_node** *A*);
  static *mp_knot do_get_value_knot*(**MP** *mp*, **mp_token_node** *A*);
#endif
  static void *do_set_value_sym*(**MP** *mp*, **mp_token_node** *A*, **mp_sym** *B*);
  static void *do_set_value_number*(**MP** *mp*, **mp_token_node** *A*, **mp_number** *B*);
  static void *do_set_value_node*(**MP** *mp*, **mp_token_node** *A*, **mp_node** *B*);
  static void *do_set_value_str*(**MP** *mp*, **mp_token_node** *A*, **mp_string** *B*);
  static void *do_set_value_knot*(**MP** *mp*, **mp_token_node** *A*, *mp_knot B*);

239.

  static mp_node *mp_get_token_node*(**MP** *mp*)
  {
    mp_node *p*;
    if (*mp*→*token_nodes*) {
      p = *mp*→*token_nodes*;
      mp→*token_nodes* = *p*→*link*;
      mp→*num_token_nodes* −−;
      p→*link* = Λ;
    }
    else {
      p = *malloc_node*(*token_node_size*);
      new_number(*p*→*data.n*);
      p→*has_number* = 1;
    }
    p→*type* = *mp_token_node_type*;
    FUNCTION_TRACE2("%p␣=␣mp_get_token_node()\n", *p*);
    return (**mp_node**) *p*;
  }

240.   **static void** *mp_free_token_node*(**MP** *mp*, **mp_node** *p*)
  {
    FUNCTION_TRACE2("mp_free_token_node(%p)\n", *p*);
    if (¬*p*) **return**;
    if (*mp*→*num_token_nodes* < *max_num_token_nodes*) {
      p→*link* = *mp*→*token_nodes*;
      mp→*token_nodes* = *p*;
      mp→*num_token_nodes* ++;
      return;
    }
    mp→*var_used* −= *token_node_size*;
    if (*mp*→*math_mode* > *mp_math_double_mode*) {
      free_number(((**mp_value_node**) *p*)→*data.n*);
    }
    xfree(*p*);
  }

241. ⟨Declarations 8⟩ +≡
  static void *mp_free_token_node*(**MP** *mp*, **mp_node** *p*);

242.   A numeric token is created by the following trivial routine.

  static mp_node *mp_new_num_tok*(**MP** *mp*, **mp_number** *v*)
  {
    mp_node *p*;   /∗ the new node ∗/
    p = *mp_get_token_node*(*mp*);
    set_value_number(*p*, *v*);
    p→*type* = *mp_known*;
    p→*name_type* = *mp_token*;
    FUNCTION_TRACE3("%p␣=␣mp_new_num_tok(%p)\n", *p*, *v*);
    return *p*;
  }

243.   A token list is a singly linked list of nodes in *mem*, where each node contains a token and a link.
Here's a subroutine that gets rid of a token list when it is no longer needed.

  static void *mp_flush_token_list*(**MP** *mp*, **mp_node** *p*)
  {
    mp_node *q*;   /∗ the node being recycled ∗/
    FUNCTION_TRACE2("mp_flush_token_list(%p)\n", *p*);
    while (*p* ≠ Λ) {
      q = *p*;
      p = *mp_link*(*p*);
      if (*mp_type*(*q*) ≡ *mp_symbol_node*) {
        mp_free_symbolic_node(*mp*, *q*);
      }
      else {
        switch (*mp_type*(*q*)) {
        case *mp_vacuous*: **case** *mp_boolean_type*: **case** *mp_known*: **break**;
        case *mp_string_type*: *delete_str_ref*(*value_str*(*q*));
          break;
        case *unknown_types*: **case** *mp_pen_type*: **case** *mp_path_type*: **case** *mp_picture_type*:
          case *mp_pair_type*: **case** *mp_color_type*: **case** *mp_cmykcolor_type*: **case** *mp_transform_type*:
          case *mp_dependent*: **case** *mp_proto_dependent*: **case** *mp_independent*:
          mp_recycle_value(*mp*, *q*);
          break;
        default: *mp_confusion*(*mp*, "token");
          ;
        }
        mp_free_token_node(*mp*, *q*);
      }
    }
  }

244.   The procedure *show_token_list*, which prints a symbolic form of the token list that starts at a given node p, illustrates these conventions. The token list being displayed should not begin with a reference count.

An additional parameter q is also given; this parameter is either NULL or it points to a node in the token list where a certain magic computation takes place that will be explained later. (Basically, q is non-NULL when we are printing the two-line context information at the time of an error message; q marks the place corresponding to where the second line should begin.)

The generation will stop, and ' `ETC.` ' will be printed, if the length of printing exceeds a given limit l; the length of printing upon entry is assumed to be a given amount called *null_tally*. (Note that *show_token_list* sometimes uses itself recursively to print variable names within a capsule.)

Unusual entries are printed in the form of all-caps tokens preceded by a space, e.g., ' `BAD`'.

⟨ Declarations 8 ⟩ +≡
    static void *mp_show_token_list*(**MP** *mp*, **mp_node** *p*, **mp_node** *q*, **integer** *l*, **integer** *null_tally*);

245.   **void** *mp_show_token_list* (**MP** *mp*, **mp_node** *p*, **mp_node** *q*, **integer** *l*, **integer** *null_tally*)
{
  quarterword *cclass*, *c*;   /∗ the *char_class* of previous and new tokens ∗/
  cclass = *percent_class*;
  mp→tally = *null_tally*;
  while ((*p* ≠ Λ) ∧ (*mp→tally* < *l*)) {
    if (*p* ≡ *q*) {
      set_trick_count ( );
    }   /∗ Display token *p* and set *c* to its class; but **return** if there are problems ∗/
    c = *letter_class*;   /∗ the default ∗/
    if (*mp_type* (*p*) ≠ *mp_symbol_node*) {   /∗ Display non-symbolic token ∗/
      if (*mp_name_type* (*p*) ≡ *mp_token*) {
        if (*mp_type* (*p*) ≡ *mp_known*) {   /∗ Display a numeric token ∗/
          if (*cclass* ≡ *digit_class*) *mp_print_char* (*mp*, *xord* ('␣'));
          if (*number_negative* (*value_number* (*p*))) {
            if (*cclass* ≡ *mp_left_bracket_class*) *mp_print_char* (*mp*, *xord* ('␣'));
            mp_print_char (*mp*, *xord* ('['));
            print_number (*value_number* (*p*));
            mp_print_char (*mp*, *xord* (']'));
            c = *mp_right_bracket_class*;
          }
          else {
            print_number (*value_number* (*p*));
            c = *digit_class*;
          }
        }
        else if (*mp_type* (*p*) ≠ *mp_string_type*) {
          mp_print (*mp*, "␣BAD");
        }
        else {
          mp_print_char (*mp*, *xord* ('"'));
          mp_print_str (*mp*, *value_str* (*p*));
          mp_print_char (*mp*, *xord* ('"'));
          c = *string_class*;
        }
      }
      else if ((*mp_name_type* (*p*) ≠ *mp_capsule*) ∨ (*mp_type* (*p*) < *mp_vacuous*) ∨ (*mp_type* (*p*) >
          mp_independent)) {
        mp_print (*mp*, "␣BAD");
      }
      else {
        mp_print_capsule (*mp*, *p*);
        c = *right_paren_class*;
      }
    }
    else {
      if (*mp_name_type* (*p*) ≡ *mp_expr_sym* ∨ *mp_name_type* (*p*) ≡ *mp_suffix_sym* ∨ *mp_name_type* (*p*) ≡
          mp_text_sym) {
        integer *r*;   /∗ temporary register ∗/
        r = *mp_sym_info* (*p*);
        if (*mp_name_type* (*p*) ≡ *mp_expr_sym*) {
          mp_print (*mp*, "(EXPR");

```
        }
        else if (mp_name_type(p) ≡ mp_suffix_sym) {
           mp_print(mp, "(SUFFIX");
        }
        else {
           mp_print(mp, "(TEXT");
        }
        mp_print_int(mp, r);
        mp_print_char(mp, xord(')'));
        c = right_paren_class;
      }
      else {
        mp_sym sr = mp_sym_sym(p);
        if (sr ≡ collective_subscript) {      /* Display a collective subscript */
           if (cclass ≡ mp_left_bracket_class) mp_print_char(mp, xord('␣'));
           mp_print(mp, "[]");
           c = mp_right_bracket_class;
        }
        else {
           mp_string rr = text(sr);
           if (rr ≡ Λ ∨ rr→str ≡ Λ) {
              mp_print(mp, "␣NONEXISTENT");
           }
           else {      /* Print string r as a symbolic token and set c to its class */
              c = (quarterword) mp→char_class[(rr→str[0])];
              if (c ≡ cclass) {
                 switch (c) {
                 case letter_class: mp_print_char(mp, xord('.'));
                    break;
                 case isolated_classes: break;
                 default: mp_print_char(mp, xord('␣'));
                    break;
                 }
              }
              mp_print_str(mp, rr);
           }
        }
      }
    }
    cclass = c;
    p = mp_link(p);
   }
   if (p ≠ Λ) mp_print(mp, "␣ETC.");
   return;
 }
```

246.   ⟨ Declarations 8 ⟩ +≡
  static void *mp_print_capsule*(**MP** *mp*, **mp_node** *p*);

247.   ⟨Declare miscellaneous procedures that were declared *forward* 247⟩ ≡
  void *mp_print_capsule*(**MP** *mp*, **mp_node** *p*)
  {
    mp_print_char(*mp*, *xord*('('));
    mp_print_exp(*mp*, *p*, 0);
    mp_print_char(*mp*, *xord*(')'));
  }

This code is used in section 1285.

248.   Macro definitions are kept in METAPOST's memory in the form of token lists that have a few extra symbolic nodes at the beginning.

  The first node contains a reference count that is used to tell when the list is no longer needed. To emphasize the fact that a reference count is present, we shall refer to the *sym_info* field of this special node as the *ref_count* field.

  The next node or nodes after the reference count serve to describe the formal parameters. They consist of zero or more parameter tokens followed by a code for the type of macro.

    /* reference count preceding a macro definition or picture header */

#define   *ref_count*(*A*)   *indep_value*(*A*)
#define   *set_ref_count*(*A*, *B*)   *set_indep_value*(*A*, *B*)
#define   *add_mac_ref*(*A*)   *set_ref_count*((*A*), *ref_count*((*A*)) + 1)
         /* make a new reference to a macro list */
#define   *decr_mac_ref*(*A*)   *set_ref_count*((*A*), *ref_count*((*A*)) − 1)
         /* remove a reference to a macro list */

⟨Types in the outer block 33⟩ +≡
  typedef enum {
    mp_general_macro,   /* preface to a macro defined with a parameter list */
    mp_primary_macro,   /* preface to a macro with a **primary** parameter */
    mp_secondary_macro,   /* preface to a macro with a **secondary** parameter */
    mp_tertiary_macro,   /* preface to a macro with a **tertiary** parameter */
    mp_expr_macro,   /* preface to a macro with an undelimited **expr** parameter */
    mp_of_macro,   /* preface to a macro with undelimited '**expr** *x* **of** *y*' parameters */
    mp_suffix_macro,   /* preface to a macro with an undelimited **suffix** parameter */
    mp_text_macro,   /* preface to a macro with an undelimited **text** parameter */
    mp_expr_param,   /* used by **expr** primitive */
    mp_suffix_param,   /* used by **suffix** primitive */
    mp_text_param   /* used by **text** primitive */
  } **mp_macro_info**;

249.   **static void** *mp_delete_mac_ref*(**MP** *mp*, **mp_node** *p*)
  {   /* *p* points to the reference count of a macro list that is losing one reference */
    if (*ref_count*(*p*) ≡ 0) *mp_flush_token_list*(*mp*, *p*);
    else *decr_mac_ref*(*p*);
  }

250.   The following subroutine displays a macro, given a pointer to its reference count.

static void mp_show_macro(**MP** mp, **mp_node** p, **mp_node** q, **integer** l)
{
  mp_node r;   /∗ temporary storage ∗/
  $p = mp\_link(p)$;   /∗ bypass the reference count ∗/
  while ($mp\_name\_type(p) \neq mp\_macro\_sym$) {
    $r = mp\_link(p)$;
    $mp\_link(p) = \Lambda$;
    $mp\_show\_token\_list(mp, p, \Lambda, l, 0)$;
    $mp\_link(p) = r$;
    $p = r$;
    if ($l > 0$) $l = l - mp{\rightarrow}tally$;
    else return;
  }   /∗ control printing of 'ETC.' ∗/
  ;
  $mp{\rightarrow}tally = 0$;
  switch ($mp\_sym\_info(p)$) {
  case $mp\_general\_macro$: $mp\_print(mp, "->")$;
    break;
  case $mp\_primary\_macro$: **case** $mp\_secondary\_macro$: **case** $mp\_tertiary\_macro$:
    $mp\_print\_char(mp, xord('<'))$;
    $mp\_print\_cmd\_mod(mp, mp\_param\_type, mp\_sym\_info(p))$;
    $mp\_print(mp, ">->")$;
    break;
  case mp_expr_macro: $mp\_print(mp, "<expr>->")$;
    break;
  case mp_of_macro: $mp\_print(mp, "<expr>of<primary>->")$;
    break;
  case mp_suffix_macro: $mp\_print(mp, "<suffix>->")$;
    break;
  case mp_text_macro: $mp\_print(mp, "<text>->")$;
    break;
  }   /∗ there are no other cases ∗/
  $mp\_show\_token\_list(mp, mp\_link(p), q, l - mp{\rightarrow}tally, 0)$;
}

251.   Data structures for variables.   The variables of METAPOST programs can be simple, like 'x', or they can combine the structural properties of arrays and records, like 'x20a.b'. A METAPOST user assigns a type to a variable like x20a.b by saying, for example, 'boolean x[]a.b'. It's time for us to study how such things are represented inside of the computer.

Each variable value occupies two consecutive words, either in a non-symbolic node called a value node, or as a non-symbolic subfield of a larger node. One of those two words is called the *value* field; it is an integer, containing either a *scaled* numeric value or the representation of some other type of quantity. (It might also be subdivided into halfwords, in which case it is referred to by other names instead of *value*.) The other word is broken into subfields called *type*, *name_type*, and *link*. The *type* field is a quarterword that specifies the variable's type, and *name_type* is a quarterword from which METAPOST can reconstruct the variable's name (sometimes by using the *link* field as well). Thus, only 1.25 words are actually devoted to the value itself; the other three-quarters of a word are overhead, but they aren't wasted because they allow METAPOST to deal with sparse arrays and to provide meaningful diagnostics.

In this section we shall be concerned only with the structural aspects of variables, not their values. Later parts of the program will change the *type* and *value* fields, but we shall treat those fields as black boxes whose contents should not be touched.

However, if the *type* field is *mp_structured*, there is no *value* field, and the second word is broken into two pointer fields called *attr_head* and *subscr_head*. Those fields point to additional nodes that contain structural information, as we shall see.

TH Note: DEK and JDH had a nice theoretical split between *value*, *attr* and *subscr* nodes, as documented above and further below. However, all three types had a bad habit of transmuting into each other in practice while pointers to them still lived on elsewhere, so using three different C structures is simply not workable. All three are now represented as a single C structure called **mp_value_node**.

There is a potential union in this structure in the interest of space saving: *subscript_* and *hashloc_* are mutually exclusive.

Actually, so are *attr_head_* + *subscr_head_* on one side and and *value_* on the other, but because of all the access macros that are used in the code base to get at values, those cannot be folded into a union (yet); this would have required creating a similar union in **mp_token_node** where it would only serve to confuse things.

Finally, *parent_* only applies in *attr* nodes (the ones that have *hashloc_*), but creating an extra substructure inside the union just for that does not save space and the extra complication in the structure is not worth the minimal extra code clarification.

#define   *attr_head*(A)   *do_get_attr_head*(*mp*, (**mp_value_node**)(A))
#define   *set_attr_head*(A, B)   *do_set_attr_head*(*mp*, (**mp_value_node**)(A), (**mp_node**)(B))
#define   *subscr_head*(A)   *do_get_subscr_head*(*mp*, (**mp_value_node**)(A))
#define   *set_subscr_head*(A, B)   *do_set_subscr_head*(*mp*, (**mp_value_node**)(A), (**mp_node**)(B))
⟨ MPlib internal header stuff 6 ⟩ +≡
  typedef struct mp_value_node_data {
    NODE_BODY;

    mp_value_data *data*;
    mp_number *subscript_*;
    mp_sym *hashloc_*;
    mp_node *parent_*;
    mp_node *attr_head_*;
    mp_node *subscr_head_*;
  } **mp_value_node_data**;

252.   **static mp_node** *do_get_attr_head*(**MP** *mp*, **mp_value_node** *A*)
  {
    assert(*A*→*type* ≡ *mp_structured*);
    FUNCTION_TRACE3("%p␣=␣get_attr_head(%p)\n", *A*→*attr_head_*, *A*);
    return *A*→*attr_head_*;
  }
  static mp_node *do_get_subscr_head*(**MP** *mp*, **mp_value_node** *A*)
  {
    assert(*A*→*type* ≡ *mp_structured*);
    FUNCTION_TRACE3("%p␣=␣get_subscr_head(%p)\n", *A*→*subscr_head_*, *A*);
    return *A*→*subscr_head_*;
  }
  static void *do_set_attr_head*(**MP** *mp*, **mp_value_node** *A*, **mp_node** *d*)
  {
    FUNCTION_TRACE4("set_attr_head(%p,%p)␣on␣line␣%d\n", (*A*), *d*, __LINE__);
    assert(*A*→*type* ≡ *mp_structured*);
    A→*attr_head_* = *d*;
  }
  static void *do_set_subscr_head*(**MP** *mp*, **mp_value_node** *A*, **mp_node** *d*)
  {
    FUNCTION_TRACE4("set_subscr_head(%p,%p)␣on␣line␣%d\n", (*A*), *d*, __LINE__);
    assert(*A*→*type* ≡ *mp_structured*);
    A→*subscr_head_* = *d*;
  }

253.   ⟨ Declarations 8 ⟩ +≡
  static mp_node *do_get_subscr_head*(**MP** *mp*, **mp_value_node** *A*);
  static mp_node *do_get_attr_head*(**MP** *mp*, **mp_value_node** *A*);
  static void *do_set_attr_head*(**MP** *mp*, **mp_value_node** *A*, **mp_node** *d*);
  static void *do_set_subscr_head*(**MP** *mp*, **mp_value_node** *A*, **mp_node** *d*);

254.   It would have been nicer to make *mp_get_value_node* return **mp_value_node** variables, but with *eqtb* as it stands that became messy: lots of typecasts. So, it returns a simple **mp_node** for now.

#define *value_node_size* **sizeof**(**struct mp_value_node_data**)

```
    static mp_node mp_get_value_node(MP mp)
    {
      mp_value_node p;
      if (mp→value_nodes) {
        p = (mp_value_node) mp→value_nodes;
        mp→value_nodes = p→link;
        mp→num_value_nodes −−;
        p→link = Λ;
      }
      else {
        p = malloc_node(value_node_size);
        new_number(p→data.n);
        new_number(p→subscript_);
        p→has_number = 2;
      }
      mp_type(p) = mp_value_node_type;
      FUNCTION_TRACE2("%p␣=␣mp_get_value_node()\n", p);
      return (mp_node) p;
    }
#if DEBUG > 1
    static void debug_dump_value_node(mp_node x)
    {
      mp_value_node qq = (mp_value_node) x;
      fprintf(stdout, "\nnode␣%p:\n", qq);
      fprintf(stdout, "␣␣type=%s\n", mp_type_string(qq→type));
      fprintf(stdout, "␣␣name_type=%d\n", qq→name_type);
      fprintf(stdout, "␣␣link=%p\n", qq→link);
      fprintf(stdout, "␣␣data.n=%d\n", qq→data.n.type);
      if (is_number(qq→data.n)) {
        fprintf(stdout, "␣␣␣␣data.n.data.val=%d\n", qq→data.n.data.val);
        fprintf(stdout, "␣␣␣␣data.n.data.dval=%f\n", qq→data.n.data.dval);
      }
      fprintf(stdout, "␣␣data.str=%p\n", qq→data.str);
      if (qq→data.str ≠ Λ) {
        fprintf(stdout, "␣␣␣␣data.str->len=%d\n", (int) qq→data.str→len);
        fprintf(stdout, "␣␣␣␣data.str->str=%s\n", qq→data.str→str);
      }
      fprintf(stdout, "␣␣data.indep.serial=%d\n␣␣data.indep.scale=%d\n", qq→data.indep.serial,
          qq→data.indep.scale);
      fprintf(stdout, "␣␣data.sym=%p\n", qq→data.sym);
      fprintf(stdout, "␣␣data.p=%p\n", qq→data.p);
      fprintf(stdout, "␣␣data.node=%p\n", qq→data.node);
      fprintf(stdout, "␣␣subscript=%d\n", qq→subscript_.type);
      if (is_number(qq→subscript_)) {
        fprintf(stdout, "␣␣␣␣subscript_.data.val=%d\n", qq→subscript_.data.val);
        fprintf(stdout, "␣␣␣␣subscript_.data.dval=%f\n", qq→subscript_.data.dval);
      }
      fprintf(stdout, "␣␣hashloc=%p\n", qq→hashloc_);
```

```
        fprintf (stdout, "␣␣parent=%p\n", qq→parent_);
        fprintf (stdout, "␣␣attr_head=%p\n", qq→attr_head_);
        fprintf (stdout, "␣␣subscr_head=%p\n\n", qq→subscr_head_);
    }
#endif
```

255.   ⟨ Declarations 8 ⟩ +≡
  static mp_node *mp_get_value_node*(**MP** *mp*);
#**if** DEBUG > 1
  static void *debug_dump_value_node*(**mp_node** *x*);
#**endif**

256.   An attribute node is three words long. Two of these words contain *type* and *value* fields as described above, and the third word contains additional information: There is an *hashloc* field, which contains the hash address of the token that names this attribute; and there's also a *parent* field, which points to the value node of *mp_structured* type at the next higher level (i.e., at the level to which this attribute is subsidiary). The *name_type* in an attribute node is '*attr*'. The *link* field points to the next attribute with the same parent; these are arranged in increasing order, so that $hashloc(mp\_link(p)) > hashloc(p)$. The final attribute node links to the constant *end_attr*, whose *hashloc* field is greater than any legal hash address. The *attr_head* in the parent points to a node whose *name_type* is *mp_structured_root*; this node represents the NULL attribute, i.e., the variable that is relevant when no attributes are attached to the parent. The *attr_head* node has the fields of either a value node, a subscript node, or an attribute node, depending on what the parent would be if it were not structured; but the subscript and attribute fields are ignored, so it effectively contains only the data of a value node. The *link* field in this special node points to an attribute node whose *hashloc* field is zero; the latter node represents a collective subscript '[]' attached to the parent, and its *link* field points to the first non-special attribute node (or to *end_attr* if there are none).

A subscript node likewise occupies three words, with *type* and *value* fields plus extra information; its *name_type* is *subscr*. In this case the third word is called the *subscript* field, which is a *scaled* integer. The *link* field points to the subscript node with the next larger subscript, if any; otherwise the *link* points to the attribute node for collective subscripts at this level. We have seen that the latter node contains an upward pointer, so that the parent can be deduced.

The *name_type* in a parent-less value node is *root*, and the *link* is the hash address of the token that names this value.

In other words, variables have a hierarchical structure that includes enough threads running around so that the program is able to move easily between siblings, parents, and children. An example should be helpful: (The reader is advised to draw a picture while reading the following description, since that will help to firm up the ideas.) Suppose that 'x' and 'x.a' and 'x[]b' and 'x5' and 'x20b' have been mentioned in a user's program, where x[]b has been declared to be of **boolean** type. Let $h(x)$, $h(a)$, and $h(b)$ be the hash addresses of x, a, and b. Then $eq\_type(h(x)) = name$ and $equiv(h(x)) = p$, where p is a non-symbolic value node with $mp\_name\_type(p) = root$ and $mp\_link(p) = h(x)$. We have $type(p) = mp\_structured$, $attr\_head(p) = q$, and $subscr\_head(p) = r$, where q points to a value node and r to a subscript node. (Are you still following this? Use a pencil to draw a diagram.) The lone variable 'x' is represented by $type(q)$ and $value(q)$; furthermore $mp\_name\_type(q) = mp\_structured\_root$ and $mp\_link(q) = q1$, where $q1$ points to an attribute node representing 'x[]'. Thus $mp\_name\_type(q1) = attr$, $hashloc(q1) = collective\_subscript = 0$, $parent(q1) = p$, $type(q1) = mp\_structured$, $attr\_head(q1) = qq$, and $subscr\_head(q1) = qq1$; qq is a three-word "attribute-as-value" node with $type(qq) = numeric\_type$ (assuming that x5 is numeric, because qq represents 'x[]' with no further attributes), $mp\_name\_type(qq) = structured\_root$, $hashloc(qq) = 0$, $parent(qq) = p$, and $mp\_link(qq) = qq1$. (Now pay attention to the next part.) Node $qq1$ is an attribute node representing 'x[][]', which has never yet occurred; its *type* field is *undefined*, and its *value* field is undefined. We have $mp\_name\_type(qq1) = attr$, $hashloc(qq1) = collective\_subscript$, $parent(qq1) = q1$, and $mp\_link(qq1) = qq2$. Since $qq2$ represents 'x[]b', $type(qq2) = mp\_unknown\_boolean$; also $hashloc(qq2) = h(b)$, $parent(qq2) = q1$, $mp\_name\_type(qq2) = attr$, $mp\_link(qq2) = end\_attr$. (Maybe colored lines will help untangle your picture.) Node r is a subscript node with *type* and *value* representing 'x5'; $mp\_name\_type(r) = subscr$, $subscript(r) = 5.0$, and $mp\_link(r) = r1$ is another subscript node. To complete the picture, see if you can guess what $mp\_link(r1)$ is; give up? It's $q1$. Furthermore $subscript(r1) = 20.0$, $mp\_name\_type(r1) = subscr$, $type(r1) = mp\_structured$, $attr\_head(r1) = qqq$, $subscr\_head(r1) = qqq1$, and we finish things off with three more nodes qqq, $qqq1$, and $qqq2$ hung onto $r1$. (Perhaps you should start again with a larger sheet of paper.) The value of variable x20b appears in node $qqq2$, as you can well imagine.

If the example in the previous paragraph doesn't make things crystal clear, a glance at some of the simpler subroutines below will reveal how things work out in practice.

The only really unusual thing about these conventions is the use of collective subscript attributes. The idea is to avoid repeating a lot of type information when many elements of an array are identical macros (for which distinct values need not be stored) or when they don't have all of the possible attributes. Branches

of the structure below collective subscript attributes do not carry actual values except for macro identifiers; branches of the structure below subscript nodes do not carry significant information in their collective subscript attributes.

#**if** DEBUG
#**define** $hashloc(A)do\_get\_hashloc$ $(mp,(\mathbf{mp\_value\_node})(A))$
#**define** $set\_hashloc(A,B)do\_set\_hashloc$ $(mp,(\mathbf{mp\_value\_node})\ A,B)$
#**define** $parent(A)do\_get\_parent$ (mp,A)
#**define** $set\_parent(A,B)do\_set\_parent$ $(mp,(\mathbf{mp\_value\_node})\ A,B)$
  static mp_sym $do\_get\_hashloc(\mathbf{MP}\ mp,\mathbf{mp\_value\_node}\ A)$
  {
     $assert((A)\rightarrow type \equiv mp\_attr\_node\_type \vee (A)\rightarrow name\_type \equiv mp\_attr);$
     return $(A)\rightarrow hashloc\_;$
  }
  static void $do\_set\_hashloc(\mathbf{MP}\ mp,\mathbf{mp\_value\_node}\ A,\mathbf{mp\_sym}\ B)$
  {
     FUNCTION_TRACE4("set_hashloc(%p,%p)␣on␣line␣%d\n",$(A),(B),$__LINE__$);$
     $assert((A)\rightarrow type \equiv mp\_attr\_node\_type \vee (A)\rightarrow name\_type \equiv mp\_attr);$
     $A\rightarrow hashloc\_ = B;$
  }
  static mp_node $do\_get\_parent(\mathbf{MP}\ mp,\mathbf{mp\_value\_node}\ A)$
  {
     $assert((A)\rightarrow type \equiv mp\_attr\_node\_type \vee (A)\rightarrow name\_type \equiv mp\_attr);$
     return $(A)\rightarrow parent\_;$ /* pointer to $mp\_structured$ variable */
  }
  static void $do\_set\_parent(\mathbf{MP}\ mp,\mathbf{mp\_value\_node}\ A,\mathbf{mp\_node}\ d)$
  {
     $assert((A)\rightarrow type \equiv mp\_attr\_node\_type \vee (A)\rightarrow name\_type \equiv mp\_attr);$
     FUNCTION_TRACE4("set_parent(%p,%p)␣on␣line␣%d\n",$(A),d,$__LINE__$);$
     $A\rightarrow parent\_ = d;$
  }
#**else**
#**define** $hashloc(A)((\mathbf{mp\_value\_node})(A))\rightarrow hashloc\_$
#**define** $set\_hashloc(A,B)((\mathbf{mp\_value\_node})(A))\rightarrow hashloc\_ = B$
#**define** $parent(A)((\mathbf{mp\_value\_node})(A))\rightarrow parent\_$
#**define** $set\_parent(A,B)((\mathbf{mp\_value\_node})(A))\rightarrow parent\_ = B$
#**endif**

257.
#**define** $mp\_free\_attr\_node(a,b)$ **do**
          {
             $assert((b)\rightarrow type \equiv mp\_attr\_node\_type \vee (b)\rightarrow name\_type \equiv mp\_attr);$
             $mp\_free\_value\_node(a,b);$
          }
          while (0)
  static mp_value_node $mp\_get\_attr\_node(\mathbf{MP}\ mp)$
  {
     $\mathbf{mp\_value\_node}\ p = (\mathbf{mp\_value\_node})\ mp\_get\_value\_node(mp);$
     $mp\_type(p) = mp\_attr\_node\_type;$
     return $p;$
  }

258.   Setting the *hashloc* field of *end_attr* to a value greater than any legal hash address is done by assigning -1 typecasted to **mp_sym**, hopefully resulting in all bits being set. On systems that support negative pointer values or where typecasting -1 does not result in all bits in a pointer being set, something else needs to be done.

⟨Initialize table entries 182⟩ +≡
  $mp \rightarrow end\_attr = (\textbf{mp\_node})\ mp\_get\_attr\_node(mp)$;
  $set\_hashloc(mp \rightarrow end\_attr, (\textbf{mp\_sym})\ -1)$;
  $set\_parent((\textbf{mp\_value\_node})\ mp \rightarrow end\_attr, \Lambda)$;

259.   ⟨Free table entries 183⟩ +≡
  $mp\_free\_attr\_node(mp, mp \rightarrow end\_attr)$;

260.
#define *collective_subscript* (**void** ∗) 0   /∗ code for the attribute '[]' ∗/
#define *subscript*(*A*) ((**mp_value_node**)(*A*))→*subscript_*
#define *set_subscript*(*A*, *B*) *do_set_subscript*(*mp*, (**mp_value_node**)(*A*), *B*)
  static void *do_set_subscript*(**MP** *mp*, **mp_value_node** *A*, **mp_number** *B*)
  {
    FUNCTION_TRACE3("set_subscript(%p,%p)\n", (*A*), (*B*));
    $assert((A) \rightarrow type \equiv mp\_subscr\_node\_type \lor (A) \rightarrow name\_type \equiv mp\_subscr)$;
    $number\_clone(A \rightarrow subscript_, B)$;   /∗ subscript of this variable ∗/
  }

261.
  static mp_value_node *mp_get_subscr_node*(**MP** *mp*)
  {
    mp_value_node $p = (\textbf{mp\_value\_node})\ mp\_get\_value\_node(mp)$;
    $mp\_type(p) = mp\_subscr\_node\_type$;
    return *p*;
  }

262.   Variables of type **pair** will have values that point to four-word nodes containing two numeric values. The first of these values has *name_type* = *mp_x_part_sector* and the second has *name_type* = *mp_y_part_sector*; the *link* in the first points back to the node whose *value* points to this four-word node.

#define *x_part*(*A*) ((*mp_pair_node*)(*A*))→*x_part_*   /∗ where the **xpart** is found in a pair node ∗/
#define *y_part*(*A*) ((*mp_pair_node*)(*A*))→*y_part_*   /∗ where the **ypart** is found in a pair node ∗/
⟨MPlib internal header stuff 6⟩ +≡
  typedef struct mp_pair_node_data {
    NODE_BODY;

    mp_node *x_part_*;
    mp_node *y_part_*;
  } **mp_pair_node_data**;
  typedef struct mp_pair_node_data ∗**mp_pair_node**;

263.

#define *pair_node_size*   **sizeof**(**struct mp_pair_node_data**)
          /∗ the number of words in a subscript node ∗/

  static mp_node *mp_get_pair_node*(**MP** *mp*)
  {
    mp_node *p*;
    if (*mp→pair_nodes*) {
       p = *mp→pair_nodes*;
       mp→pair_nodes = *p→link*;
       mp→num_pair_nodes −−;
       p→link = Λ;
    }
    else {
       p = *malloc_node*(*pair_node_size*);
    }
    mp_type(*p*) = *mp_pair_node_type*;
    FUNCTION_TRACE2("get_pair_node():␣%p\n", *p*);
    return (**mp_node**) *p*;
  }

264.   ⟨ Declarations 8 ⟩ +≡
  void *mp_free_pair_node*(**MP** *mp*, **mp_node** *p*);

265.   **void** *mp_free_pair_node*(**MP** *mp*, **mp_node** *p*)
  {
    FUNCTION_TRACE2("mp_free_pair_node(%p)\n", *p*);
    if (¬*p*) **return**;
    if (*mp→num_pair_nodes* < *max_num_pair_nodes*) {
       p→link = *mp→pair_nodes*;
       mp→pair_nodes = *p*;
       mp→num_pair_nodes ++;
       return;
    }
    mp→var_used −= *pair_node_size*;
    xfree(*p*);
  }

266.   If $type(p) = mp\_pair\_type$ or if $value(p) = \Lambda$, the procedure call $init\_pair\_node(p)$ will allocate a pair node for p. The individual parts of such nodes are initially of type $mp\_independent$.

> **static void** $mp\_init\_pair\_node(\mathbf{MP}\ mp, \mathbf{mp\_node}\ p)$
> {
>   **mp_node** q;   /* the new node */
>
> $mp\_type(p) = mp\_pair\_type$;
> $q = mp\_get\_pair\_node(mp)$;
> $y\_part(q) = mp\_get\_value\_node(mp)$;
> $mp\_new\_indep(mp, y\_part(q))$;   /* sets $type(q)$ and $value(q)$ */
> $mp\_name\_type(y\_part(q)) = (\mathbf{quarterword})(mp\_y\_part\_sector)$;
> $mp\_link(y\_part(q)) = p$;
> $x\_part(q) = mp\_get\_value\_node(mp)$;
> $mp\_new\_indep(mp, x\_part(q))$;   /* sets $type(q)$ and $value(q)$ */
> $mp\_name\_type(x\_part(q)) = (\mathbf{quarterword})(mp\_x\_part\_sector)$;
> $mp\_link(x\_part(q)) = p$;
> $set\_value\_node(p, q)$;
> }

267.   Variables of type **transform** are similar, but in this case their *value* points to a 12-word node containing six values, identified by x_part_sector, y_part_sector, $mp\_xx\_part\_sector$, $mp\_xy\_part\_sector$, $mp\_yx\_part\_sector$, ■ and $mp\_yy\_part\_sector$.

#define $tx\_part(A)$ $((mp\_transform\_node)(A))\rightarrow tx\_part\_$
            /* where the **xpart** is found in a transform node */
#define $ty\_part(A)$ $((mp\_transform\_node)(A))\rightarrow ty\_part\_$
            /* where the **ypart** is found in a transform node */
#define $xx\_part(A)$ $((mp\_transform\_node)(A))\rightarrow xx\_part\_$
            /* where the **xxpart** is found in a transform node */
#define $xy\_part(A)$ $((mp\_transform\_node)(A))\rightarrow xy\_part\_$
            /* where the **xypart** is found in a transform node */
#define $yx\_part(A)$ $((mp\_transform\_node)(A))\rightarrow yx\_part\_$
            /* where the **yxpart** is found in a transform node */
#define $yy\_part(A)$ $((mp\_transform\_node)(A))\rightarrow yy\_part\_$
            /* where the **yypart** is found in a transform node */

⟨MPlib internal header stuff 6⟩ +≡
  typedef struct mp_transform_node_data {
    NODE_BODY;

    mp_node $tx\_part\_$;
    mp_node $ty\_part\_$;
    mp_node $xx\_part\_$;
    mp_node $yx\_part\_$;
    mp_node $xy\_part\_$;
    mp_node $yy\_part\_$;
  } **mp_transform_node_data**;
  typedef struct mp_transform_node_data *$\mathbf{mp\_transform\_node}$;

268.

#define *transform_node_size* **sizeof**(**struct mp_transform_node_data**)
          /∗ the number of words in a subscript node ∗/

  static mp_node *mp_get_transform_node*(**MP** *mp*)
  {
    mp_transform_node *p* = (**mp_transform_node**) *malloc_node*(*transform_node_size*);
    mp_type(*p*) = *mp_transform_node_type*;
    return (**mp_node**) *p*;
  }

269.   **static void** *mp_init_transform_node*(**MP** *mp*, **mp_node** *p*)
  {
    mp_node *q*;   /∗ the new node ∗/
    mp_type(*p*) = *mp_transform_type*;
    q = *mp_get_transform_node*(*mp*);   /∗ big node ∗/
    yy_part(*q*) = *mp_get_value_node*(*mp*);
    mp_new_indep(*mp*, *yy_part*(*q*));   /∗ sets *type*(*q*) and *value*(*q*) ∗/
    mp_name_type(*yy_part*(*q*)) = (**quarterword**)(*mp_yy_part_sector*);
    mp_link(*yy_part*(*q*)) = *p*;
    yx_part(*q*) = *mp_get_value_node*(*mp*);
    mp_new_indep(*mp*, *yx_part*(*q*));   /∗ sets *type*(*q*) and *value*(*q*) ∗/
    mp_name_type(*yx_part*(*q*)) = (**quarterword**)(*mp_yx_part_sector*);
    mp_link(*yx_part*(*q*)) = *p*;
    xy_part(*q*) = *mp_get_value_node*(*mp*);
    mp_new_indep(*mp*, *xy_part*(*q*));   /∗ sets *type*(*q*) and *value*(*q*) ∗/
    mp_name_type(*xy_part*(*q*)) = (**quarterword**)(*mp_xy_part_sector*);
    mp_link(*xy_part*(*q*)) = *p*;
    xx_part(*q*) = *mp_get_value_node*(*mp*);
    mp_new_indep(*mp*, *xx_part*(*q*));   /∗ sets *type*(*q*) and *value*(*q*) ∗/
    mp_name_type(*xx_part*(*q*)) = (**quarterword**)(*mp_xx_part_sector*);
    mp_link(*xx_part*(*q*)) = *p*;
    ty_part(*q*) = *mp_get_value_node*(*mp*);
    mp_new_indep(*mp*, *ty_part*(*q*));   /∗ sets *type*(*q*) and *value*(*q*) ∗/
    mp_name_type(*ty_part*(*q*)) = (**quarterword**)(*mp_y_part_sector*);
    mp_link(*ty_part*(*q*)) = *p*;
    tx_part(*q*) = *mp_get_value_node*(*mp*);
    mp_new_indep(*mp*, *tx_part*(*q*));   /∗ sets *type*(*q*) and *value*(*q*) ∗/
    mp_name_type(*tx_part*(*q*)) = (**quarterword**)(*mp_x_part_sector*);
    mp_link(*tx_part*(*q*)) = *p*;
    set_value_node(*p*, *q*);
  }

270.   Variables of type **color** have 3 values in 6 words identified by $mp\_red\_part\_sector$, $mp\_green\_part\_sector$,∎ and $mp\_blue\_part\_sector$.

#define $red\_part(A)$ $((\textbf{mp\_color\_node})(A)) \rightarrow red\_part\_$
          /∗ where the **redpart** is found in a color node ∗/
#define $green\_part(A)$ $((\textbf{mp\_color\_node})(A)) \rightarrow green\_part\_$
          /∗ where the **greenpart** is found in a color node ∗/
#define $blue\_part(A)$ $((\textbf{mp\_color\_node})(A)) \rightarrow blue\_part\_$
          /∗ where the **bluepart** is found in a color node ∗/
#define $grey\_part(A)$ $red\_part(A)$ /∗ where the **greypart** is found in a color node ∗/

⟨MPlib internal header stuff 6⟩ +≡
  typedef struct mp_color_node_data {
    NODE_BODY;

    mp_node $red\_part\_$;
    mp_node $green\_part\_$;
    mp_node $blue\_part\_$;
  } **mp_color_node_data**;
  typedef struct mp_color_node_data ∗**mp_color_node**;

271.

#define $color\_node\_size$ **sizeof**(**struct mp_color_node_data**)
          /∗ the number of words in a subscript node ∗/

  static mp_node $mp\_get\_color\_node(\textbf{MP}\ mp)$
  {
    mp_color_node $p = (\textbf{mp\_color\_node})\ malloc\_node(color\_node\_size)$;

    $mp\_type(p) = mp\_color\_node\_type$;
    $p \rightarrow link = \Lambda$;
    return (**mp_node**) p;
  }

272.
  static void $mp\_init\_color\_node(\textbf{MP}\ mp, \textbf{mp\_node}\ p)$
  {
    mp_node q;   /∗ the new node ∗/

    $mp\_type(p) = mp\_color\_type$;
    $q = mp\_get\_color\_node(mp)$;   /∗ big node ∗/
    $blue\_part(q) = mp\_get\_value\_node(mp)$;
    $mp\_new\_indep(mp, blue\_part(q))$;   /∗ sets $type(q)$ and $value(q)$ ∗/
    $mp\_name\_type(blue\_part(q)) = (\textbf{quarterword})(mp\_blue\_part\_sector)$;
    $mp\_link(blue\_part(q)) = p$;
    $green\_part(q) = mp\_get\_value\_node(mp)$;
    $mp\_new\_indep(mp, green\_part(q))$;   /∗ sets $type(q)$ and $value(q)$ ∗/
    $mp\_name\_type(y\_part(q)) = (\textbf{quarterword})(mp\_green\_part\_sector)$;
    $mp\_link(green\_part(q)) = p$;
    $red\_part(q) = mp\_get\_value\_node(mp)$;
    $mp\_new\_indep(mp, red\_part(q))$;   /∗ sets $type(q)$ and $value(q)$ ∗/
    $mp\_name\_type(red\_part(q)) = (\textbf{quarterword})(mp\_red\_part\_sector)$;
    $mp\_link(red\_part(q)) = p$;
    $set\_value\_node(p, q)$;
  }

273.   Finally, variables of type *cmykcolor*.

#define *cyan_part*(*A*)   ((**mp_cmykcolor_node**)(*A*))⃗*cyan_part*_
            /∗ where the **cyanpart** is found in a color node ∗/
#define *magenta_part*(*A*)   ((**mp_cmykcolor_node**)(*A*))⃗*magenta_part*_
            /∗ where the **magentapart** is found in a color node ∗/
#define *yellow_part*(*A*)   ((**mp_cmykcolor_node**)(*A*))⃗*yellow_part*_
            /∗ where the **yellowpart** is found in a color node ∗/
#define *black_part*(*A*)   ((**mp_cmykcolor_node**)(*A*))⃗*black_part*_
            /∗ where the **blackpart** is found in a color node ∗/

⟨ MPlib internal header stuff 6 ⟩ +≡
  typedef struct mp_cmykcolor_node_data {
    NODE_BODY;

    mp_node *cyan_part*_;
    mp_node *magenta_part*_;
    mp_node *yellow_part*_;
    mp_node *black_part*_;
  } **mp_cmykcolor_node_data**;
  typedef struct mp_cmykcolor_node_data ∗**mp_cmykcolor_node**;

274.

#define *cmykcolor_node_size*   **sizeof**(**struct mp_cmykcolor_node_data**)
            /∗ the number of words in a subscript node ∗/
  static mp_node *mp_get_cmykcolor_node*(**MP** *mp*)
  {
    mp_cmykcolor_node *p* = (**mp_cmykcolor_node**) *malloc_node*(*cmykcolor_node_size*);

    mp_type(*p*) = *mp_cmykcolor_node_type*;
    p⃗*link* = Λ;
    return (**mp_node**) *p*;
  }

275.

```
static void mp_init_cmykcolor_node(MP mp, mp_node p)
{
  mp_node q;      /* the new node */
  mp_type(p) = mp_cmykcolor_type;
  q = mp_get_cmykcolor_node(mp);      /* big node */
  black_part(q) = mp_get_value_node(mp);
  mp_new_indep(mp, black_part(q));      /* sets type(q) and value(q) */
  mp_name_type(black_part(q)) = (quarterword)(mp_black_part_sector);
  mp_link(black_part(q)) = p;
  yellow_part(q) = mp_get_value_node(mp);
  mp_new_indep(mp, yellow_part(q));      /* sets type(q) and value(q) */
  mp_name_type(yellow_part(q)) = (quarterword)(mp_yellow_part_sector);
  mp_link(yellow_part(q)) = p;
  magenta_part(q) = mp_get_value_node(mp);
  mp_new_indep(mp, magenta_part(q));      /* sets type(q) and value(q) */
  mp_name_type(magenta_part(q)) = (quarterword)(mp_magenta_part_sector);
  mp_link(magenta_part(q)) = p;
  cyan_part(q) = mp_get_value_node(mp);
  mp_new_indep(mp, cyan_part(q));      /* sets type(q) and value(q) */
  mp_name_type(cyan_part(q)) = (quarterword)(mp_cyan_part_sector);
  mp_link(cyan_part(q)) = p;
  set_value_node(p, q);
}
```

276.   When an entire structured variable is saved, the *root* indication is temporarily replaced by *saved_root*.

Some variables have no name; they just are used for temporary storage while expressions are being evaluated. We call them *capsules*.

277.   The *id_transform* function creates a capsule for the identity transformation.

static mp_node *mp_id_transform*(**MP** *mp*)
{
   mp_node *p*, *q*;   /∗ list manipulation registers ∗/
   p = *mp_get_value_node*(*mp*);
   mp_name_type(*p*) = *mp_capsule*;
   set_value_number(*p*, *zero_t*);   /∗ todo: this was *null* ∗/
   mp_init_transform_node(*mp*, *p*);
   q = *value_node*(*p*);
   mp_type(*tx_part*(*q*)) = *mp_known*;
   set_value_number(*tx_part*(*q*), *zero_t*);
   mp_type(*ty_part*(*q*)) = *mp_known*;
   set_value_number(*ty_part*(*q*), *zero_t*);
   mp_type(*xy_part*(*q*)) = *mp_known*;
   set_value_number(*xy_part*(*q*), *zero_t*);
   mp_type(*yx_part*(*q*)) = *mp_known*;
   set_value_number(*yx_part*(*q*), *zero_t*);
   mp_type(*xx_part*(*q*)) = *mp_known*;
   set_value_number(*xx_part*(*q*), *unity_t*);
   mp_type(*yy_part*(*q*)) = *mp_known*;
   set_value_number(*yy_part*(*q*), *unity_t*);
   return *p*;
}

278.   Tokens are of type *tag_token* when they first appear, but they point to Λ until they are first used as the root of a variable. The following subroutine establishes the root node on such grand occasions.

static void *mp_new_root*(**MP** *mp*, **mp_sym** *x*)
{
   mp_node *p*;   /∗ the new node ∗/
   p = *mp_get_value_node*(*mp*);
   mp_type(*p*) = *mp_undefined*;
   mp_name_type(*p*) = *mp_root*;
   set_value_sym(*p*, *x*);
   set_equiv_node(*x*, *p*);
}

279.   These conventions for variable representation are illustrated by the *print_variable_name* routine, which displays the full name of a variable given only a pointer to its value.

⟨ Declarations 8 ⟩ +≡
   static void *mp_print_variable_name*(**MP** *mp*, **mp_node** *p*);

280.   **void** $mp\_print\_variable\_name(\mathbf{MP}\ mp, \mathbf{mp\_node}\ p)$
  {
    mp_node q;   /∗ a token list that will name the variable's suffix ∗/
    mp_node r;   /∗ temporary for token list creation ∗/

    while $(mp\_name\_type(p) \geq mp\_x\_part\_sector)$ {
      switch $(mp\_name\_type(p))$ {
      case $mp\_x\_part\_sector$: $mp\_print(mp, \texttt{"xpart\textvisiblespace"})$;
        break;
      case $mp\_y\_part\_sector$: $mp\_print(mp, \texttt{"ypart\textvisiblespace"})$;
        break;
      case $mp\_xx\_part\_sector$: $mp\_print(mp, \texttt{"xxpart\textvisiblespace"})$;
        break;
      case $mp\_xy\_part\_sector$: $mp\_print(mp, \texttt{"xypart\textvisiblespace"})$;
        break;
      case $mp\_yx\_part\_sector$: $mp\_print(mp, \texttt{"yxpart\textvisiblespace"})$;
        break;
      case $mp\_yy\_part\_sector$: $mp\_print(mp, \texttt{"yypart\textvisiblespace"})$;
        break;
      case $mp\_red\_part\_sector$: $mp\_print(mp, \texttt{"redpart\textvisiblespace"})$;
        break;
      case $mp\_green\_part\_sector$: $mp\_print(mp, \texttt{"greenpart\textvisiblespace"})$;
        break;
      case $mp\_blue\_part\_sector$: $mp\_print(mp, \texttt{"bluepart\textvisiblespace"})$;
        break;
      case $mp\_cyan\_part\_sector$: $mp\_print(mp, \texttt{"cyanpart\textvisiblespace"})$;
        break;
      case $mp\_magenta\_part\_sector$: $mp\_print(mp, \texttt{"magentapart\textvisiblespace"})$;
        break;
      case $mp\_yellow\_part\_sector$: $mp\_print(mp, \texttt{"yellowpart\textvisiblespace"})$;
        break;
      case $mp\_black\_part\_sector$: $mp\_print(mp, \texttt{"blackpart\textvisiblespace"})$;
        break;
      case $mp\_grey\_part\_sector$: $mp\_print(mp, \texttt{"greypart\textvisiblespace"})$;
        break;
      case $mp\_capsule$: $mp\_printf(mp, \texttt{"\%\%CAPSULE\%p"}, p)$;
        return;
        break;   /∗ this is to please the compiler: the remaining cases are operation codes ∗/
      default: **break**;
      }
      $p = mp\_link(p)$;
    }
    $q = \Lambda$;
    while $(mp\_name\_type(p) > mp\_saved\_root)$ {
        /∗ Ascend one level, pushing a token onto list q and replacing p by its parent ∗/
      if $(mp\_name\_type(p) \equiv mp\_subscr)$ {
        $r = mp\_new\_num\_tok(mp, subscript(p))$;
        do {
          $p = mp\_link(p)$;
        } **while** $(mp\_name\_type(p) \neq mp\_attr)$;
      }
      else if $(mp\_name\_type(p) \equiv mp\_structured\_root)$ {
        $p = mp\_link(p)$;

```
      goto FOUND;
   }
   else {
      if (mp_name_type(p) ≠ mp_attr) mp_confusion(mp, "var");
      r = mp_get_symbolic_node(mp);
      set_mp_sym_sym(r, hashloc(p));        /∗ the hash address ∗/
   }
   set_mp_link(r, q);
   q = r;
FOUND: p = parent((mp_value_node) p);
}      /∗ now link(p) is the hash address of p, and name_type(p) is either root or saved_root. Have to
         prepend a token to q for show_token_list. ∗/
r = mp_get_symbolic_node(mp);
set_mp_sym_sym(r, value_sym(p));
mp_link(r) = q;
if (mp_name_type(p) ≡ mp_saved_root) mp_print(mp, "(SAVED)");
mp_show_token_list(mp, r, Λ, max_integer, mp→tally);
mp_flush_token_list(mp, r);
}
```

281.   The *interesting* function returns *true* if a given variable is not in a capsule, or if the user wants to
trace capsules.

```
static boolean mp_interesting(MP mp, mp_node p)
{
  mp_name_type_type t;      /* a name_type */
  if (number_positive(internal_value(mp_tracing_capsules))) {
    return true;
  }
  else {
    t = mp_name_type(p);
    if (t ≥ mp_x_part_sector ∧ t ≠ mp_capsule) {
      mp_node tt = value_node(mp_link(p));

      switch (t) {
      case mp_x_part_sector: t = mp_name_type(x_part(tt));
        break;
      case mp_y_part_sector: t = mp_name_type(y_part(tt));
        break;
      case mp_xx_part_sector: t = mp_name_type(xx_part(tt));
        break;
      case mp_xy_part_sector: t = mp_name_type(xy_part(tt));
        break;
      case mp_yx_part_sector: t = mp_name_type(yx_part(tt));
        break;
      case mp_yy_part_sector: t = mp_name_type(yy_part(tt));
        break;
      case mp_red_part_sector: t = mp_name_type(red_part(tt));
        break;
      case mp_green_part_sector: t = mp_name_type(green_part(tt));
        break;
      case mp_blue_part_sector: t = mp_name_type(blue_part(tt));
        break;
      case mp_cyan_part_sector: t = mp_name_type(cyan_part(tt));
        break;
      case mp_magenta_part_sector: t = mp_name_type(magenta_part(tt));
        break;
      case mp_yellow_part_sector: t = mp_name_type(yellow_part(tt));
        break;
      case mp_black_part_sector: t = mp_name_type(black_part(tt));
        break;
      case mp_grey_part_sector: t = mp_name_type(grey_part(tt));
        break;
      default: break;
      }
    }
  }
  return (t ≠ mp_capsule);
}
```

282.   Now here is a subroutine that converts an unstructured type into an equivalent structured type, by inserting a *mp_structured* node that is capable of growing. This operation is done only when $mp\_name\_type(p) =$ ▮ *root*, *subscr*, or *attr*.

The procedure returns a pointer to the new node that has taken node p's place in the structure. Node p itself does not move, nor are its *value* or *type* fields changed in any way.

> **static mp_node** *mp_new_structure*(**MP** *mp*, **mp_node** *p*)
> {
>   **mp_node** *q*, *r* = Λ;   /∗ list manipulation registers ∗/
>   **mp_sym** *qq* = Λ;
>   **switch** (*mp_name_type*(*p*)) {
>   **case** *mp_root*:
>   {
>   *qq* = *value_sym*(*p*);
>   *r* = *mp_get_value_node*(*mp*);
>   *set_equiv_node*(*qq*, *r*);
>   }
>   **break**;
>   **case** *mp_subscr*:   /∗ Link a new subscript node *r* in place of node *p* ∗/
>   {
>   **mp_node** *q_new*;
>
>   *q* = *p*;
>   **do** {
>   *q* = *mp_link*(*q*);
>   } **while** (*mp_name_type*(*q*) ≠ *mp_attr*);
>   *q* = *parent*((**mp_value_node**) *q*);
>   *r* = *mp*→*temp_head*;
>   *set_mp_link*(*r*, *subscr_head*(*q*));
>   **do** {
>   *q_new* = *r*;
>   *r* = *mp_link*(*r*);
>   } **while** (*r* ≠ *p*);
>   *r* = (**mp_node**) *mp_get_subscr_node*(*mp*);
>   **if** (*q_new* ≡ *mp*→*temp_head*) {
>   *set_subscr_head*(*q*, *r*);
>   }
>   **else** {
>   *set_mp_link*(*q_new*, *r*);
>   }
>   *set_subscript*(*r*, *subscript*(*p*));
>   }
>   **break**;
>   **case** *mp_attr*:   /∗ Link a new attribute node *r* in place of node *p* ∗/   /∗ If the attribute is
>   *collective_subscript*, there are two pointers to node *p*, so we must change both of them. ∗/
>   {
>   **mp_value_node** *rr*;
>
>   *q* = *parent*((**mp_value_node**) *p*);
>   *r* = *attr_head*(*q*);
>   **do** {
>   *q* = *r*;
>   *r* = *mp_link*(*r*);
>   } **while** (*r* ≠ *p*);

```
      rr = mp_get_attr_node(mp);
      r = (mp_node) rr;
      set_mp_link(q, (mp_node) rr);
      set_hashloc(rr, hashloc(p));
      set_parent(rr, parent((mp_value_node) p));
      if (hashloc(p) ≡ collective_subscript) {
         q = mp→temp_head;
         set_mp_link(q, subscr_head(parent((mp_value_node) p)));
         while (mp_link(q) ≠ p)  q = mp_link(q);
         if (q ≡ mp→temp_head)  set_subscr_head(parent((mp_value_node) p), (mp_node) rr);
         else  set_mp_link(q, (mp_node) rr);
      }
   }
   break;
   default: mp_confusion(mp, "struct");
      break;
   }
   set_mp_link(r, mp_link(p));
   set_value_sym(r, value_sym(p));
   mp_type(r) = mp_structured;
   mp_name_type(r) = mp_name_type(p);
   set_attr_head(r, p);
   mp_name_type(p) = mp_structured_root;
   {
      mp_value_node qqr = mp_get_attr_node(mp);

      set_mp_link(p, (mp_node) qqr);
      set_subscr_head(r, (mp_node) qqr);
      set_parent(qqr, r);
      mp_type(qqr) = mp_undefined;
      mp_name_type(qqr) = mp_attr;
      set_mp_link(qqr, mp→end_attr);
      set_hashloc(qqr, collective_subscript);
   }
   return r;
}
```

283.   The *find_variable* routine is given a pointer t to a nonempty token list of suffixes; it returns a pointer to the corresponding non-symbolic value. For example, if t points to token **x** followed by a numeric token containing the value 7, *find_variable* finds where the value of **x7** is stored in memory. This may seem a simple task, and it usually is, except when **x7** has never been referenced before. Indeed, **x** may never have even been subscripted before; complexities arise with respect to updating the collective subscript information.

If a macro type is detected anywhere along path t, or if the first item on t isn't a *tag_token*, the value Λ is returned. Otherwise p will be a non-NULL pointer to a node such that *undefined* $<$ *type*(p) $<$ *mp_structured*.

```
static mp_node mp_find_variable(MP mp, mp_node t)
{
   mp_node p, q, r, s;       /* nodes in the "value" line */
   mp_sym p_sym;
   mp_node pp, qq, rr, ss;       /* nodes in the "collective" line */
   ;
   p_sym = mp_sym_sym(t);
   t = mp_link(t);
   if ((eq_type(p_sym) % mp_outer_tag) ≠ mp_tag_token) return Λ;
   if (equiv_node(p_sym) ≡ Λ) mp_new_root(mp, p_sym);
   p = equiv_node(p_sym);
   pp = p;
   while (t ≠ Λ) {       /* Make sure that both nodes p and pp are of mp_structured type */
         /* Although pp and p begin together, they diverge when a subscript occurs; pp stays in the
            collective line while p goes through actual subscript values. */
      if (mp_type(pp) ≠ mp_structured) {
         if (mp_type(pp) > mp_structured) return Λ;
         ss = mp_new_structure(mp, pp);
         if (p ≡ pp) p = ss;
         pp = ss;
      }     /* now type(pp) = mp_structured */
      if (mp_type(p) ≠ mp_structured) {     /* it cannot be > mp_structured */
         p = mp_new_structure(mp, p);     /* now type(p) = mp_structured */
      }
   }
   if (mp_type(t) ≠ mp_symbol_node) {     /* Descend one level for the subscript value(t) */
         /* We want this part of the program to be reasonably fast, in case there are lots of subscripts
            at the same level of the data structure. Therefore we store an "infinite" value in the word
            that appears at the end of the subscript list, even though that word isn't part of a subscript
            node. */
      mp_number nn, save_subscript;       /* temporary storage */
      new_number(nn);
      new_number(save_subscript);
      number_clone(nn, value_number(t));
      pp = mp_link(attr_head(pp));       /* now hashloc(pp) = collective_subscript */
      q = mp_link(attr_head(p));
      number_clone(save_subscript, subscript(q));
      set_number_to_inf(subscript(q));
      s = mp→temp_head;
      set_mp_link(s, subscr_head(p));
      do {
         r = s;
         s = mp_link(s);
      } while (number_greater(nn, subscript(s)));
      if (number_equal(nn, subscript(s))) {
```

```
      p = s;
    }
    else {
      mp_value_node p1 = mp_get_subscr_node(mp);

      if (r ≡ mp→temp_head) set_subscr_head(p, (mp_node) p1);
      else set_mp_link(r, (mp_node) p1);
      set_mp_link(p1, s);
      number_clone(subscript(p1), nn);
      mp_name_type(p1) = mp_subscr;
      mp_type(p1) = mp_undefined;
      p = (mp_node) p1;
    }
    number_clone(subscript(q), save_subscript);
    free_number(save_subscript);
    free_number(nn);
  }
  else {      /* Descend one level for the attribute mp_sym_info(t) */
    mp_sym nn1 = mp_sym_sym(t);

    ss = attr_head(pp);
    do {
      rr = ss;
      ss = mp_link(ss);
    } while (nn1 > hashloc(ss));
    if (nn1 < hashloc(ss)) {
      qq = (mp_node) mp_get_attr_node(mp);
      set_mp_link(rr, qq);
      set_mp_link(qq, ss);
      set_hashloc(qq, nn1);
      mp_name_type(qq) = mp_attr;
      mp_type(qq) = mp_undefined;
      set_parent((mp_value_node) qq, pp);
      ss = qq;
    }
    if (p ≡ pp) {
      p = ss;
      pp = ss;
    }
    else {
      pp = ss;
      s = attr_head(p);
      do {
        r = s;
        s = mp_link(s);
      } while (nn1 > hashloc(s));
      if (nn1 ≡ hashloc(s)) {
        p = s;
      }
      else {
        q = (mp_node) mp_get_attr_node(mp);
        set_mp_link(r, q);
        set_mp_link(q, s);
        set_hashloc(q, nn1);
```

$mp\_name\_type(q) = mp\_attr;$
$mp\_type(q) = mp\_undefined;$
$set\_parent((\textbf{mp\_value\_node})\ q, p);$
$p = q;$
    }
  }
}
$t = mp\_link(t);$
}
if $(mp\_type(pp) \geq mp\_structured)$ {
  if $(mp\_type(pp) \equiv mp\_structured)$ $pp = attr\_head(pp);$
  else return $\Lambda;$
}
if $(mp\_type(p) \equiv mp\_structured)$ $p = attr\_head(p);$
if $(mp\_type(p) \equiv mp\_undefined)$ {
  if $(mp\_type(pp) \equiv mp\_undefined)$ {
    $mp\_type(pp) = mp\_numeric\_type;$
    $set\_value\_number(pp, zero\_t);$
  }
  $mp\_type(p) = mp\_type(pp);$
  $set\_value\_number(p, zero\_t);$
}
return $p;$
}

284.   Variables lose their former values when they appear in a type declaration, or when they are defined to be macros or **let** equal to something else. A subroutine will be defined later that recycles the storage associated with any particular *type* or *value*; our goal now is to study a higher level process called *flush_variable*, which selectively frees parts of a variable structure.

This routine has some complexity because of examples such as 'numeric x[]a[]b' which recycles all variables of the form x[i]a[j]b (and no others), while 'vardef x[]a[]=...' discards all variables of the form x[i]a[j] followed by an arbitrary suffix, except for the collective node x[]a[] itself. The obvious way to handle such examples is to use recursion; so that's what we do.

Parameter p points to the root information of the variable; parameter t points to a list of symbolic nodes that represent suffixes, with *info* = *collective_subscript* for subscripts.

⟨ Declarations 8 ⟩ +≡
  void $mp\_flush\_cur\_exp(\textbf{MP}\ mp, \textbf{mp\_value}\ v);$

285.   **static void** $mp\_flush\_variable$(**MP** mp, **mp_node** p, **mp_node** t, **boolean** $discard\_suffixes$)
```
  {
    mp_node q, r = Λ;      /* list manipulation */
    mp_sym n;       /* attribute to match */
    while (t ≠ Λ) {
      if (mp_type(p) ≠ mp_structured) {
        return;
      }
      n = mp_sym_sym(t);
      t = mp_link(t);
      if (n ≡ collective_subscript) {
        q = subscr_head(p);
        while (mp_name_type(q) ≡ mp_subscr) {
          mp_flush_variable(mp, q, t, discard_suffixes);
          if (t ≡ Λ) {
            if (mp_type(q) ≡ mp_structured) {
              r = q;
            }
            else {
              if (r ≡ Λ) set_subscr_head(p, mp_link(q));
              else  set_mp_link(r, mp_link(q));
              mp_free_value_node(mp, q);
            }
          }
          else {
            r = q;
          }
          q = (r ≡ Λ ? subscr_head(p) : mp_link(r));
        }
      }
      p = attr_head(p);
      do {
        p = mp_link(p);
      } while (hashloc(p) < n);
      if (hashloc(p) ≠ n) {
        return;
      }
    }
    if (discard_suffixes) {
      mp_flush_below_variable(mp, p);
    }
    else {
      if (mp_type(p) ≡ mp_structured) {
        p = attr_head(p);
      }
      mp_recycle_value(mp, p);
    }
  }
```

286.   The next procedure is simpler; it wipes out everything but p itself, which becomes undefined.

⟨ Declarations 8 ⟩ +≡
   static void $mp\_flush\_below\_variable$(**MP** mp, **mp_node** p);

287.   **void** $mp\_flush\_below\_variable$(**MP** mp, **mp_node** p)
  {
     mp_node q, r;   /∗ list manipulation registers ∗/
     FUNCTION_TRACE2("mp_flush_below_variable(%p)\n", p);
     if $(mp\_type(p) \neq mp\_structured)$ {
        $mp\_recycle\_value(mp, p)$;   /∗ this sets $type(p) = undefined$ ∗/
     }
     else {
        $q = subscr\_head(p)$;
        while $(mp\_name\_type(q) \equiv mp\_subscr)$ {
           $mp\_flush\_below\_variable(mp, q)$;
           $r = q$;
           $q = mp\_link(q)$;
           $mp\_free\_value\_node(mp, r)$;
        }
        $r = attr\_head(p)$;
        $q = mp\_link(r)$;
        $mp\_recycle\_value(mp, r)$;
        $mp\_free\_value\_node(mp, r)$;
        do {
           $mp\_flush\_below\_variable(mp, q)$;
           $r = q$;
           $q = mp\_link(q)$;
           $mp\_free\_value\_node(mp, r)$;
        } **while** $(q \neq mp\text{→}end\_attr)$;
        $mp\_type(p) = mp\_undefined$;
     }
  }

288.   Just before assigning a new value to a variable, we will recycle the old value and make the old value undefined. The *und_type* routine determines what type of undefined value should be given, based on the current type before recycling.

> **static quarterword** *mp_und_type*(**MP** *mp*, **mp_node** *p*)
> {
>   (**void**) *mp*;
>   **switch** (*mp_type*(*p*)) {
>   **case** *mp_vacuous*: **return** *mp_undefined*;
>   **case** *mp_boolean_type*: **case** *mp_unknown_boolean*: **return** *mp_unknown_boolean*;
>   **case** *mp_string_type*: **case** *mp_unknown_string*: **return** *mp_unknown_string*;
>   **case** *mp_pen_type*: **case** *mp_unknown_pen*: **return** *mp_unknown_pen*;
>   **case** *mp_path_type*: **case** *mp_unknown_path*: **return** *mp_unknown_path*;
>   **case** *mp_picture_type*: **case** *mp_unknown_picture*: **return** *mp_unknown_picture*;
>   **case** *mp_transform_type*: **case** *mp_color_type*: **case** *mp_cmykcolor_type*: **case** *mp_pair_type*:
>   **case** *mp_numeric_type*: **return** *mp_type*(*p*);
>   **case** *mp_known*: **case** *mp_dependent*: **case** *mp_proto_dependent*: **case** *mp_independent*:
>   **return** *mp_numeric_type*;
>   **default**:   /* there are no other valid cases, but please the compiler */
>   **return** 0;
>   }
>   **return** 0;
> }

289.   The *clear_symbol* routine is used when we want to redefine the equivalent of a symbolic token. It must remove any variable structure or macro definition that is currently attached to that symbol. If the *saving* parameter is true, a subsidiary structure is saved instead of destroyed.

> **static void** *mp_clear_symbol*(**MP** *mp*, **mp_sym** *p*, **boolean** *saving*)
> {
>   **mp_node** *q*;   /* *equiv*(*p*) */
>   FUNCTION_TRACE3("mp_clear_symbol(%p,%d)\n", *p*, *saving*);
>   *q* = *equiv_node*(*p*);
>   **switch** (*eq_type*(*p*) % *mp_outer_tag*) {
>   **case** *mp_defined_macro*: **case** *mp_secondary_primary_macro*: **case** *mp_tertiary_secondary_macro*:
>   **case** *mp_expression_tertiary_macro*:
>   **if** (¬*saving*) *mp_delete_mac_ref*(*mp*, *q*);
>   **break**;
>   **case** *mp_tag_token*:
>   **if** (*q* ≠ Λ) {
>   **if** (*saving*) {
>   *mp_name_type*(*q*) = *mp_saved_root*;
>   }
>   **else** {
>   *mp_flush_below_variable*(*mp*, *q*);
>   *mp_free_value_node*(*mp*, *q*);
>   }
>   }
>   **break**;
>   **default**: **break**;
>   }
>   *set_equiv*(*p*, *mp*→*frozen_undefined*→*v.data.indep.serial*);
>   *set_eq_type*(*p*, *mp*→*frozen_undefined*→*type*);
> }

290.   Saving and restoring equivalents.   The nested structure given by **begingroup** and **endgroup** allows *eqtb* entries to be saved and restored, so that temporary changes can be made without difficulty. When the user requests a current value to be saved, METAPOST puts that value into its "save stack." An appearance of **endgroup** ultimately causes the old values to be removed from the save stack and put back in their former places.

The save stack is a linked list containing three kinds of entries, distinguished by their *type* fields. If *p* points to a saved item, then

$p \rightarrow type = 0$ stands for a group boundary; each **begingroup** contributes such an item to the save stack and each **endgroup** cuts back the stack until the most recent such entry has been removed.

$p \rightarrow type = mp\_normal\_sym$ means that $p \rightarrow value$ holds the former contents of $eqtb[q]$ (saved in the *knot* field of the value, which is otherwise unused for variables). Such save stack entries are generated by **save** commands.

$p \rightarrow type = mp\_internal\_sym$ means that $p \rightarrow value$ is a **mp_internal** to be restored to internal parameter number *q* (saved in the *serial* field of the value, which is otherwise unused for internals). Such entries are generated by **interim** commands.

The global variable *save_ptr* points to the top item on the save stack.

⟨ Types in the outer block 33 ⟩ +≡
  typedef struct mp_save_data {
    quarterword *type*;
    mp_internal *value*;
    struct mp_save_data *∗link*;
  } **mp_save_data**;

291.   ⟨ Global variables 14 ⟩ +≡
  mp_save_data *∗save_ptr*;   /∗ the most recently saved item ∗/

292.   ⟨ Set initial values of key variables 38 ⟩ +≡
  $mp \rightarrow save\_ptr = \Lambda$;

293.   Saving a boundary item
  static void *mp_save_boundary*(**MP** *mp*)
  {
    mp_save_data *∗p*;   /∗ temporary register ∗/
    FUNCTION_TRACE1("mp_save_boundary␣()\n");
    $p = xmalloc(1, \textbf{sizeof}(\textbf{mp\_save\_data}))$;
    $p \rightarrow type = 0$;
    $p \rightarrow link = mp \rightarrow save\_ptr$;
    $mp \rightarrow save\_ptr = p$;
  }

294.   The *save_variable* routine is given a hash address q; it salts this address in the save stack, together with its current equivalent, then makes token q behave as though it were brand new.

Nothing is stacked when *save_ptr* $= \Lambda$, however; there's no way to remove things from the stack when the program is not inside a group, so there's no point in wasting the space.

static void *mp_save_variable*(**MP** *mp*, **mp_sym** *q*)
{
  mp_save_data *∗p*;   /∗ temporary register ∗/
  FUNCTION_TRACE2("mp_save_variable␣(%p)\n", *q*);
  if (*mp*→*save_ptr* ≠ Λ) {
    p = *xmalloc*(1, **sizeof**(**mp_save_data**));
    p→*type* = *mp_normal_sym*;
    p→*link* = *mp*→*save_ptr*;
    p→*value.v.data.indep.scale* = *eq_type*(*q*);
    p→*value.v.data.indep.serial* = *equiv*(*q*);
    p→*value.v.data.node* = *equiv_node*(*q*);
    p→*value.v.data.p* = (*mp_knot*)*q*;
    mp→*save_ptr* = *p*;
  }
  mp_clear_symbol(*mp*, *q*, (*mp*→*save_ptr* ≠ Λ));
}

static void *mp_unsave_variable*(**MP** *mp*)
{
  mp_sym *q* = (**mp_sym**) *mp*→*save_ptr*→*value.v.data.p*;
  if (*number_positive*(*internal_value*(*mp_tracing_restores*))) {
    mp_begin_diagnostic(*mp*);
    mp_print_nl(*mp*, "{restoring␣");
    mp_print_text(*q*);
    mp_print_char(*mp*, *xord*('}'));
    mp_end_diagnostic(*mp*, *false*);
  }
  mp_clear_symbol(*mp*, *q*, *false*);
  set_eq_type(*q*, *mp*→*save_ptr*→*value.v.data.indep.scale*);
  set_equiv(*q*, *mp*→*save_ptr*→*value.v.data.indep.serial*);
  q→*v.data.node* = *mp*→*save_ptr*→*value.v.data.node*;
  if (*eq_type*(*q*) % *mp_outer_tag* ≡ *mp_tag_token*) {
    mp_node *pp* = *q*→*v.data.node*;
    if (*pp* ≠ Λ) *mp_name_type*(*pp*) = *mp_root*;
  }
}

295.   Similarly, *save_internal* is given the location q of an internal quantity like *mp_tracing_pens*. It creates a save stack entry of the third kind.

```
static void mp_save_internal(MP mp, halfword q)
{
    mp_save_data *p;       /* new item for the save stack */
    FUNCTION_TRACE2("mp_save_internal␣(%d)\n", q);
    if (mp→save_ptr ≠ Λ) {
        p = xmalloc(1, sizeof(mp_save_data));
        p→type = mp_internal_sym;
        p→link = mp→save_ptr;
        p→value = mp→internal[q];
        p→value.v.data.indep.serial = q;
        new_number(p→value.v.data.n);
        number_clone(p→value.v.data.n, mp→internal[q].v.data.n);
        mp→save_ptr = p;
    }
}
static void mp_unsave_internal(MP mp)
{
    halfword q = mp→save_ptr→value.v.data.indep.serial;
    mp_internal saved = mp→save_ptr→value;
    if (number_positive(internal_value(mp_tracing_restores))) {
        mp_begin_diagnostic(mp);
        mp_print_nl(mp, "{restoring␣");
        mp_print(mp, internal_name(q));
        mp_print_char(mp, xord('='));
        if (internal_type(q) ≡ mp_known) {
            print_number(saved.v.data.n);
        }
        else if (internal_type(q) ≡ mp_string_type) {
            char *s = mp_str(mp, saved.v.data.str);
            mp_print(mp, s);
        }
        else {
            mp_confusion(mp, "internal_restore");
        }
        mp_print_char(mp, xord('}'));
        mp_end_diagnostic(mp, false);
    }
    free_number(mp→internal[q].v.data.n);
    mp→internal[q] = saved;
}
```

296.   At the end of a group, the *unsave* routine restores all of the saved equivalents in reverse order. This
routine will be called only when there is at least one boundary item on the save stack.

```
static void mp_unsave(MP mp)
{
  mp_save_data *p;      /* saved item */
  FUNCTION_TRACE1("mp_unsave␣()\n");
  while (mp→save_ptr→type ≠ 0) {
    if (mp→save_ptr→type ≡ mp_internal_sym) {
      mp_unsave_internal(mp);
    }
    else {
      mp_unsave_variable(mp);
    }
    p = mp→save_ptr→link;
    xfree(mp→save_ptr);
    mp→save_ptr = p;
  }
  p = mp→save_ptr→link;
  xfree(mp→save_ptr);
  mp→save_ptr = p;
}
```

297.   Data structures for paths.   When a METAPOST user specifies a path, METAPOST will create a
list of knots and control points for the associated cubic spline curves. If the knots are z_0, z_1, $\ldots$, z_n, there
are control points z_k^+ and z_{k+1}^- such that the cubic splines between knots z_k and z_{k+1} are defined by Bézier's
formula

$$z(t) = B(z_k, z_k^+, z_{k+1}^-, z_{k+1}; t)$$
$$= (1-t)^3 z_k + 3(1-t)^2 t z_k^+ + 3(1-t)t^2 z_{k+1}^- + t^3 z_{k+1}$$

for $0 \le t \le 1$.

There is a 8-word node for each knot z_k, containing one word of control information and six words for
the x and y coordinates of z_k^- and z_k and z_k^+. The control information appears in the *mp_left_type* and
mp_right_type fields, which each occupy a quarter of the first word in the node; they specify properties of
the curve as it enters and leaves the knot. There's also a halfword *link* field, which points to the following
knot, and a final supplementary word (of which only a quarter is used).

If the path is a closed contour, knots 0 and n are identical; i.e., the *link* in knot $n-1$ points to knot 0. But
if the path is not closed, the *mp_left_type* of knot 0 and the *mp_right_type* of knot n are equal to *endpoint*.
In the latter case the *link* in knot n points to knot 0, and the control points z_0^- and z_n^+ are not used.

#define *mp_next_knot*(A) (A)→*next*   /∗ the next knot in this list ∗/
#define *mp_left_type*(A) (A)→*data.types.left_type*   /∗ characterizes the path entering this knot ∗/
#define *mp_right_type*(A) (A)→*data.types.right_type*   /∗ characterizes the path leaving this knot ∗/
#define *mp_prev_knot*(A) (A)→*data.prev*   /∗ the previous knot in this list (only for pens) ∗/
#define *mp_knot_info*(A) (A)→*data.info*   /∗ temporary info, used during splitting ∗/

⟨ Exported types 15 ⟩ +≡
  typedef struct *mp_knot_data* ∗**mp_knot**;
  typedef struct mp_knot_data {
    mp_number *x_coord*;   /∗ the x coordinate of this knot ∗/
    mp_number *y_coord*;   /∗ the y coordinate of this knot ∗/
    mp_number *left_x*;   /∗ the x coordinate of previous control point ∗/
    mp_number *left_y*;   /∗ the y coordinate of previous control point ∗/
    mp_number *right_x*;   /∗ the x coordinate of next control point ∗/
    mp_number *right_y*;   /∗ the y coordinate of next control point ∗/
    mp_knot *next*;
    union {
      struct {
        unsigned short *left_type*;
        unsigned short *right_type*;
      } *types*;
      mp_knot *prev*;
      signed int *info*;
    } *data*;
    unsigned char *originator*;
  } **mp_knot_data**;

298.

#define $mp\_gr\_next\_knot(A)$ $(A)→next$ /∗ the next knot in this list ∗/

⟨ Exported types 15 ⟩ +≡

   typedef struct $mp\_gr\_knot\_data$ ∗**mp_gr_knot**;

   typedef struct mp_gr_knot_data {

     double x_coord;

     double y_coord;

     double $left\_x$;

     double $left\_y$;

     double $right\_x$;

     double $right\_y$;

     mp_gr_knot $next$;

     union {

       struct {

         unsigned short $left\_type$;

         unsigned short $right\_type$;

       } $types$;

       mp_gr_knot $prev$;

       signed int $info$;

     } $data$;

     unsigned char $originator$;

   } **mp_gr_knot_data**;

299.   ⟨ MPlib header stuff 201 ⟩ +≡

   enum mp_knot_type {

     $mp\_endpoint = 0,$ /∗ mp_left_type at path beginning and mp_right_type at path end ∗/

     $mp\_explicit,$ /∗ mp_left_type or mp_right_type when control points are known ∗/

     $mp\_given,$ /∗ mp_left_type or mp_right_type when a direction is given ∗/

     $mp\_curl,$ /∗ mp_left_type or mp_right_type when a curl is desired ∗/

     $mp\_open,$ /∗ mp_left_type or mp_right_type when METAPOST should choose the direction ∗/

     mp_end_cycle

   };

300.   Before the Bézier control points have been calculated, the memory space they will ultimately occupy is taken up by information that can be used to compute them. There are four cases:

- If $mp\_right\_type = mp\_open$, the curve should leave the knot in the same direction it entered; META-POST will figure out a suitable direction.

- If $mp\_right\_type = mp\_curl$, the curve should leave the knot in a direction depending on the angle at which it enters the next knot and on the curl parameter stored in $right\_curl$.

- If $mp\_right\_type = mp\_given$, the curve should leave the knot in a nonzero direction stored as an *angle* in $right\_given$.

- If $mp\_right\_type = mp\_explicit$, the Bézier control point for leaving this knot has already been computed; it is in the mp_right_x and mp_right_y fields.

The rules for mp_left_type are similar, but they refer to the curve entering the knot, and to *left* fields instead of *right* fields.

Non-**explicit** control points will be chosen based on "tension" parameters in the $left\_tension$ and $right\_tension$▮ fields. The '**atleast**' option is represented by negative tension values.

For example, the METAPOST path specification

```
z0..z1..tension atleast 1..{curl 2}z2..z3{-1,-2}..tension 3 and 4..p,
```

where p is the path 'z4..controls z45 and z54..z5', will be represented by the six knots

mp_left_type	*left* info	x_coord, y_coord	mp_right_type	*right* info
endpoint	—, —	x_0, y_0	*curl*	$1.0, 1.0$
open	—, 1.0	x_1, y_1	*open*	—, -1.0
curl	$2.0, -1.0$	x_2, y_2	*curl*	$2.0, 1.0$
given	$d, 1.0$	x_3, y_3	*given*	$d, 3.0$
open	—, 4.0	x_4, y_4	**explicit**	x_{45}, y_{45}
explicit	x_{54}, y_{54}	x_5, y_5	*endpoint*	—, —

Here d is the *angle* obtained by calling $n\_arg(-unity, -two)$. Of course, this example is more complicated than anything a normal user would ever write.

These types must satisfy certain restrictions because of the form of METAPOST's path syntax: (i) *open* type never appears in the same node together with *endpoint*, *given*, or *curl*. (ii) The mp_right_type of a node is **explicit** if and only if the mp_left_type of the following node is **explicit**. (iii) *endpoint* types occur only at the ends, as mentioned above.

#define $left\_curl$ $left\_x$ /∗ curl information when entering this knot ∗/
#define $left\_given$ $left\_x$ /∗ given direction when entering this knot ∗/
#define $left\_tension$ $left\_y$ /∗ tension information when entering this knot ∗/
#define $right\_curl$ $right\_x$ /∗ curl information when leaving this knot ∗/
#define $right\_given$ $right\_x$ /∗ given direction when leaving this knot ∗/
#define $right\_tension$ $right\_y$ /∗ tension information when leaving this knot ∗/

301.   Knots can be user-supplied, or they can be created by program code, like the *split_cubic* function, or *copy_path*. The distinction is needed for the cleanup routine that runs after *split_cubic*, because it should only delete knots it has previously inserted, and never anything that was user-supplied. In order to be able to differentiate one knot from another, we will set *originator*(*p*): = *mp_metapost_user* when it appeared in the actual metapost program, and *originator*(*p*): = *mp_program_code* in all other cases.

#define   *mp_originator*(*A*)   (*A*)→*originator*   /∗ the creator of this knot ∗/

⟨ Exported types 15 ⟩ +≡
  enum mp_knot_originator {
    mp_program_code = 0,   /∗ not created by a user ∗/
    mp_metapost_user   /∗ created by a user ∗/
  };

302.   Here is a routine that prints a given knot list in symbolic form. It illustrates the conventions discussed above, and checks for anomalies that might arise while METAPOST is being debugged.

⟨ Declarations 8 ⟩ +≡
  static void *mp_pr_path*(**MP** *mp*, **mp_knot** *h*);

303.   **void** *mp_pr_path*(**MP** *mp*, **mp_knot** *h*)
  {
    mp_knot *p*, *q*;   /∗ for list traversal ∗/
    p = *h*;
    do {
      q = *mp_next_knot*(*p*);
      if ((*p* ≡ Λ) ∨ (*q* ≡ Λ)) {
        mp_print_nl(*mp*, "???");
        return;   /∗ this won't happen ∗/
      }
      ⟨ Print information for adjacent knots *p* and *q* 304 ⟩;
    DONE1: *p* = *q*;
      if (*p* ∧ ((*p* ≠ *h*) ∨ (*mp_left_type*(*h*) ≠ *mp_endpoint*))) {
        ⟨ Print two dots, followed by *given* or *curl* if present 305 ⟩;
      }
    } **while** (*p* ≠ *h*);
    if (*mp_left_type*(*h*) ≠ *mp_endpoint*) *mp_print*(*mp*, "cycle");
  }

304.   ⟨Print information for adjacent knots p and q 304⟩ ≡
  $mp\_print\_two(mp, p\text{-}x\_coord, p\text{-}y\_coord)$;
  switch $(mp\_right\_type(p))$ {
  case $mp\_endpoint$:
    if $(mp\_left\_type(p) \equiv mp\_open)$ $mp\_print(mp, "\{open?\}")$;   /∗ can't happen ∗/
    ;
    if $((mp\_left\_type(q) \neq mp\_endpoint) \vee (q \neq h))$ $q = \Lambda$;   /∗ force an error ∗/
    goto DONE1;
    break;
  case $mp\_explicit$: ⟨Print control points between p and q, then **goto** *done1* 307⟩;
    break;
  case mp_open: ⟨Print information for a curve that begins *open* 308⟩;
    break;
  case mp_curl: **case** mp_given: ⟨Print information for a curve that begins *curl* or *given* 309⟩;
    break;
  default: $mp\_print(mp, "???")$;   /∗ can't happen ∗/
    ;
    break;
  }
  if $(mp\_left\_type(q) \leq mp\_explicit)$ {
    $mp\_print(mp, "..control?")$;   /∗ can't happen ∗/
  }
  else if $((\neg number\_equal(p\text{-}right\_tension, unity\_t)) \vee (\neg number\_equal(q\text{-}left\_tension, unity\_t)))$ {
    ⟨Print tension between p and q 306⟩;
  }
This code is used in section 303.

305.   Since n_sin_cos produces *fraction* results, which we will print as if they were *scaled*, the magnitude of a *given* direction vector will be 4096.

⟨ Print two dots, followed by *given* or *curl* if present 305 ⟩ ≡

```
{
    mp_number n_sin, n_cos;

    new_fraction(n_sin);
    new_fraction(n_cos);
    mp_print_nl(mp, "␣..");
    if (mp_left_type(p) ≡ mp_given) {
        n_sin_cos(p⃗left_given, n_cos, n_sin);
        mp_print_char(mp, xord('{'));
        print_number(n_cos);
        mp_print_char(mp, xord(','));
        print_number(n_sin);
        mp_print_char(mp, xord('}'));
    }
    else if (mp_left_type(p) ≡ mp_curl) {
        mp_print(mp, "{curl␣");
        print_number(p⃗left_curl);
        mp_print_char(mp, xord('}'));
    }
    free_number(n_sin);
    free_number(n_cos);
}
```

This code is used in section 303.

306.   ⟨ Print tension between p and q 306 ⟩ ≡

```
{
    mp_number v1;

    new_number(v1);
    mp_print(mp, "..tension␣");
    if (number_negative(p⃗right_tension)) mp_print(mp, "atleast");
    number_clone(v1, p⃗right_tension);
    number_abs(v1);
    print_number(v1);
    if (¬number_equal(p⃗right_tension, q⃗left_tension)) {
        mp_print(mp, "␣and␣");
        if (number_negative(q⃗left_tension)) mp_print(mp, "atleast");
        number_clone(v1, p⃗left_tension);
        number_abs(v1);
        print_number(v1);
    }
    free_number(v1);
}
```

This code is used in section 304.

307.   ⟨Print control points between p and q, then **goto** *done1* 307⟩ ≡
```
{
    mp_print(mp, "..controls␣");
    mp_print_two(mp, p→right_x, p→right_y);
    mp_print(mp, "␣and␣");
    if (mp_left_type(q) ≠ mp_explicit) {
        mp_print(mp, "??");       /* can't happen */
    }
    else {
        mp_print_two(mp, q→left_x, q→left_y);
    }
    goto DONE1;
}
```
This code is used in section 304.

308.   ⟨Print information for a curve that begins *open* 308⟩ ≡
```
if ((mp_left_type(p) ≠ mp_explicit) ∧ (mp_left_type(p) ≠ mp_open)) {
    mp_print(mp, "{open?}");       /* can't happen */
}
```
This code is used in section 304.

309.   A curl of 1 is shown explicitly, so that the user sees clearly that METAPOST's default curl is present.

⟨Print information for a curve that begins *curl* or *given* 309⟩ ≡
```
{
    if (mp_left_type(p) ≡ mp_open) mp_print(mp, "??");       /* can't happen */
    ;
    if (mp_right_type(p) ≡ mp_curl) {
        mp_print(mp, "{curl␣");
        print_number(p→right_curl);
    }
    else {
        mp_number n_sin, n_cos;

        new_fraction(n_sin);
        new_fraction(n_cos);
        n_sin_cos(p→right_given, n_cos, n_sin);
        mp_print_char(mp, xord('{'));
        print_number(n_cos);
        mp_print_char(mp, xord(','));
        print_number(n_sin);
        free_number(n_sin);
        free_number(n_cos);
    }
    mp_print_char(mp, xord('}'));
}
```
This code is used in section 304.

310.   It is convenient to have another version of *pr_path* that prints the path as a diagnostic message.

⟨Declarations 8⟩ +≡
```
    static void mp_print_path(MP mp, mp_knot h, const char *s, boolean nuline);
```

311.   **void** mp_print_path(**MP** mp, **mp_knot** h, **const char** $*s$, **boolean** $nuline$)
{
   $mp\_print\_diagnostic(mp, \texttt{"Path"}, s, nuline)$;
   $mp\_print\_ln(mp)$;
   ;
   $mp\_pr\_path(mp, h)$;
   $mp\_end\_diagnostic(mp, true)$;
}

312.   ⟨ Declarations 8 ⟩ +≡
  static mp_knot mp_new_knot(**MP** mp);

313.   **static mp_knot** mp_new_knot(**MP** mp)
{
  mp_knot q;
  if (mp→$knot\_nodes$) {
    $q = mp$→$knot\_nodes$;
    mp→$knot\_nodes = q$→$next$;
    mp→num_knot_nodes −−;
  }
  else {
    $q = mp\_xmalloc(mp, 1, \textbf{sizeof}(\textbf{struct mp\_knot\_data}))$;
  }
  $memset(q, 0, \textbf{sizeof}(\textbf{struct mp\_knot\_data}))$;
  $new\_number(q$→$x\_coord)$;
  $new\_number(q$→$y\_coord)$;
  $new\_number(q$→$left\_x)$;
  $new\_number(q$→$left\_y)$;
  $new\_number(q$→$right\_x)$;
  $new\_number(q$→$right\_y)$;
  return q;
}

314.   ⟨ Declarations 8 ⟩ +≡
  static mp_gr_knot $mp\_gr\_new\_knot$(**MP** mp);

315.   **static mp_gr_knot** $mp\_gr\_new\_knot$(**MP** mp)
{
  mp_gr_knot $q = mp\_xmalloc(mp, 1, \textbf{sizeof}(\textbf{struct mp\_gr\_knot\_data}))$;
  return q;
}

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

```
static mp_knot mp_copy_knot(MP mp, mp_knot p)
{
  mp_knot q;
  if (mp→knot_nodes) {
    q = mp→knot_nodes;
    mp→knot_nodes = q→next;
    mp→num_knot_nodes−−;
  }
  else {
    q = mp_xmalloc(mp, 1, sizeof(struct mp_knot_data));
  }
  memcpy(q, p, sizeof(struct mp_knot_data));
  if (mp→math_mode > mp_math_double_mode) {
    new_number(q→x_coord);
    new_number(q→y_coord);
    new_number(q→left_x);
    new_number(q→left_y);
    new_number(q→right_x);
    new_number(q→right_y);
    number_clone(q→x_coord, p→x_coord);
    number_clone(q→y_coord, p→y_coord);
    number_clone(q→left_x, p→left_x);
    number_clone(q→left_y, p→left_y);
    number_clone(q→right_x, p→right_x);
    number_clone(q→right_y, p→right_y);
  }
  mp_next_knot(q) = Λ;
  return q;
}
```

317.   If we want to export a knot node, we can say *export_knot*:

```
static mp_gr_knot mp_export_knot(MP mp, mp_knot p)
{
  mp_gr_knot q;      /* the copy */
  q = mp_gr_new_knot(mp);
  q→x_coord = number_to_double(p→x_coord);
  q→y_coord = number_to_double(p→y_coord);
  q→left_x = number_to_double(p→left_x);
  q→left_y = number_to_double(p→left_y);
  q→right_x = number_to_double(p→right_x);
  q→right_y = number_to_double(p→right_y);
  q→data.types.left_type = mp_left_type(p);
  q→data.types.right_type = mp_left_type(p);
  q→data.info = mp_knot_info(p);
  mp_gr_next_knot(q) = Λ;
  return q;
}
```

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

```
static mp_knot mp_copy_path(MP mp, mp_knot p)
{
    mp_knot q, pp, qq;      /* for list manipulation */
    if (p ≡ Λ) return Λ;
    q = mp_copy_knot(mp, p);
    qq = q;
    pp = mp_next_knot(p);
    while (pp ≠ p) {
        mp_next_knot(qq) = mp_copy_knot(mp, pp);
        qq = mp_next_knot(qq);
        pp = mp_next_knot(pp);
    }
    mp_next_knot(qq) = q;
    return q;
}
```

319.   The *export_path* routine makes a clone of a given path and converts the *value*s therein to **double**s.

```
static mp_gr_knot mp_export_path(MP mp, mp_knot p)
{
    mp_knot pp;      /* for list manipulation */
    mp_gr_knot q, qq;
    if (p ≡ Λ) return Λ;
    q = mp_export_knot(mp, p);
    qq = q;
    pp = mp_next_knot(p);
    while (pp ≠ p) {
        mp_gr_next_knot(qq) = mp_export_knot(mp, pp);
        qq = mp_gr_next_knot(qq);
        pp = mp_next_knot(pp);
    }
    mp_gr_next_knot(qq) = q;
    return q;
}
```

320.   If we want to import a knot node, we can say *import_knot*:

  static mp_knot *mp_import_knot*(**MP** *mp*, **mp_gr_knot** *p*)

  {

    mp_knot *q*;   /∗ the copy ∗/

    q = *mp_new_knot*(*mp*);

    set_number_from_double(*q⃗x_coord*, *p⃗x_coord*);

    set_number_from_double(*q⃗y_coord*, *p⃗y_coord*);

    set_number_from_double(*q⃗left_x*, *p⃗left_x*);

    set_number_from_double(*q⃗left_y*, *p⃗left_y*);

    set_number_from_double(*q⃗right_x*, *p⃗right_x*);

    set_number_from_double(*q⃗right_y*, *p⃗right_y*);

    mp_left_type(*q*) = *p⃗data*.*types*.*left_type*;

    mp_left_type(*q*) = *p⃗data*.*types*.*right_type*;

    mp_knot_info(*q*) = *p⃗data*.*info*;

    mp_next_knot(*q*) = Λ;

    return *q*;

  }

321.   The *import_path* routine makes a clone of a given path and converts the *value*s therein to *scaled*s.

  static mp_knot *mp_import_path*(**MP** *mp*, **mp_gr_knot** *p*)

  {

    mp_gr_knot *pp*;   /∗ for list manipulation ∗/

    mp_knot *q*, *qq*;

    if (*p* ≡ Λ) **return** Λ;

    q = *mp_import_knot*(*mp*, *p*);

    qq = *q*;

    pp = *mp_gr_next_knot*(*p*);

    while (*pp* ≠ *p*) {

      mp_next_knot(*qq*) = *mp_import_knot*(*mp*, *pp*);

      qq = *mp_next_knot*(*qq*);

      pp = *mp_gr_next_knot*(*pp*);

    }

    mp_next_knot(*qq*) = *q*;

    return *q*;

  }

322.   Just before *ship_out*, knot lists are exported for printing.

323.   The *export_knot_list* routine therefore also makes a clone of a given path.

static mp_gr_knot *mp_export_knot_list*(**MP** *mp*, **mp_knot** *p*)
{
   mp_gr_knot *q*;   /∗ the exported copy ∗/
   if ($p \equiv \Lambda$) **return** Λ;
   $q = mp\_export\_path(mp, p)$;
   return *q*;
}
static mp_knot *mp_import_knot_list*(**MP** *mp*, **mp_gr_knot** *q*)
{
   mp_knot *p*;   /∗ the imported copy ∗/
   if ($q \equiv \Lambda$) **return** Λ;
   $p = mp\_import\_path(mp, q)$;
   return *p*;
}

324.   Similarly, there's a way to copy the *reverse* of a path. This procedure returns a pointer to the first node of the copy, if the path is a cycle, but to the final node of a non-cyclic copy. The global variable *path_tail* will point to the final node of the original path; this trick makes it easier to implement '**doublepath**'.

   All node types are assumed to be *endpoint* or **explicit** only.

static mp_knot *mp_htap_ypoc*(**MP** *mp*, **mp_knot** *p*)
{
   mp_knot *q*, *pp*, *qq*, *rr*;   /∗ for list manipulation ∗/
   $q = mp\_new\_knot(mp)$;   /∗ this will correspond to *p* ∗/
   $qq = q$;
   $pp = p$;
   while (1) {
      $mp\_right\_type(qq) = mp\_left\_type(pp)$;
      $mp\_left\_type(qq) = mp\_right\_type(pp)$;
      $number\_clone(qq{\rightarrow}x\_coord, pp{\rightarrow}x\_coord)$;
      $number\_clone(qq{\rightarrow}y\_coord, pp{\rightarrow}y\_coord)$;
      $number\_clone(qq{\rightarrow}right\_x, pp{\rightarrow}left\_x)$;
      $number\_clone(qq{\rightarrow}right\_y, pp{\rightarrow}left\_y)$;
      $number\_clone(qq{\rightarrow}left\_x, pp{\rightarrow}right\_x)$;
      $number\_clone(qq{\rightarrow}left\_y, pp{\rightarrow}right\_y)$;
      $mp\_originator(qq) = mp\_originator(pp)$;
      if ($mp\_next\_knot(pp) \equiv p$) {
         $mp\_next\_knot(q) = qq$;
         $mp{\rightarrow}path\_tail = pp$;
         return *q*;
      }
      $rr = mp\_new\_knot(mp)$;
      $mp\_next\_knot(rr) = qq$;
      $qq = rr$;
      $pp = mp\_next\_knot(pp)$;
   }
}

325.   ⟨Global variables 14⟩ +≡
   mp_knot *path_tail*;   /∗ the node that links to the beginning of a path ∗/

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

⟨ Declarations 8 ⟩ +≡

   static void *mp_toss_knot_list*(**MP** *mp*, **mp_knot** *p*);

   static void *mp_toss_knot*(**MP** *mp*, **mp_knot** *p*);

   static void *mp_free_knot*(**MP** *mp*, **mp_knot** *p*);

327.   **void** $mp\_free\_knot(\textbf{MP } mp, \textbf{mp\_knot } q)$
{
   $free\_number(q{\rightarrow}x\_coord);$
   $free\_number(q{\rightarrow}y\_coord);$
   $free\_number(q{\rightarrow}left\_x);$
   $free\_number(q{\rightarrow}left\_y);$
   $free\_number(q{\rightarrow}right\_x);$
   $free\_number(q{\rightarrow}right\_y);$
   $mp\_xfree(q);$
}
void $mp\_toss\_knot(\textbf{MP } mp, \textbf{mp\_knot } q)$
{
   if $(mp{\rightarrow}num\_knot\_nodes < max\_num\_knot\_nodes)$ {
     $q{\rightarrow}next = mp{\rightarrow}knot\_nodes;$
     $mp{\rightarrow}knot\_nodes = q;$
     $mp{\rightarrow}num\_knot\_nodes \mathbin{+}{+};$
     return;
   }
   if $(mp{\rightarrow}math\_mode > mp\_math\_double\_mode)$ {
     $mp\_free\_knot(mp, q);$
   }
   else {
     $mp\_xfree(q);$
   }
}
void $mp\_toss\_knot\_list(\textbf{MP } mp, \textbf{mp\_knot } p)$
{
   mp_knot $q;$ /∗ the node being freed ∗/
   mp_knot $r;$ /∗ the next node ∗/
   if $(p \equiv \Lambda)$ **return**;
   $q = p;$
   if $(mp{\rightarrow}math\_mode > mp\_math\_double\_mode)$ {
     do {
       $r = mp\_next\_knot(q);$
       $mp\_toss\_knot(mp, q);$
       $q = r;$
     } **while** $(q \neq p);$
   }
   else {
     do {
       $r = mp\_next\_knot(q);$
       if $(mp{\rightarrow}num\_knot\_nodes < max\_num\_knot\_nodes)$ {
         $q{\rightarrow}next = mp{\rightarrow}knot\_nodes;$
         $mp{\rightarrow}knot\_nodes = q;$
         $mp{\rightarrow}num\_knot\_nodes \mathbin{+}{+};$
       }
       else {
         $mp\_xfree(q);$
       }
       $q = r;$
     } **while** $(q \neq p);$

```
    }
  }
```

328.   Choosing control points.   Now we must actually delve into one of METAPOST's more difficult routines, the *make_choices* procedure that chooses angles and control points for the splines of a curve when the user has not specified them explicitly. The parameter to *make_choices* points to a list of knots and path information, as described above.

A path decomposes into independent segments at "breakpoint" knots, which are knots whose left and right angles are both prespecified in some way (i.e., their *mp_left_type* and *mp_right_type* aren't both open).

> **void** *mp_make_choices*(**MP** *mp*, **mp_knot** *knots*)
> {
>   **mp_knot** *h*;   /∗ the first breakpoint ∗/
>   **mp_knot** *p*, *q*;   /∗ consecutive breakpoints being processed ∗/
>   ⟨Other local variables for *make_choices* 342⟩;
>   FUNCTION_TRACE1("make_choices()\n");
>   *check_arith*( );   /∗ make sure that *arith_error* = *false* ∗/
>   **if** (*number_positive*(*internal_value*(*mp_tracing_choices*)))
>   *mp_print_path*(*mp*, *knots*, ",␣before␣choices", *true*);
>   ⟨If consecutive knots are equal, join them explicitly 331⟩;
>   ⟨Find the first breakpoint, *h*, on the path; insert an artificial breakpoint if the path is an unbroken
>   cycle 332⟩;
>   *p* = *h*;
>   **do** {
>   ⟨Fill in the control points between *p* and the next breakpoint, then advance *p* to that breakpoint 333⟩;
>   } **while** (*p* ≠ *h*);
>   **if** (*number_positive*(*internal_value*(*mp_tracing_choices*)))
>   *mp_print_path*(*mp*, *knots*, ",␣after␣choices", *true*);
>   **if** (*mp*→*arith_error*) {
>   ⟨Report an unexpected problem during the choice-making 330⟩;
>   }
> }

329.   ⟨Internal library declarations 10⟩ +≡
> **void** *mp_make_choices*(**MP** *mp*, **mp_knot** *knots*);

330.   ⟨Report an unexpected problem during the choice-making 330⟩ ≡
> {
>   **const char** ∗*hlp*[ ] = {"The␣path␣that␣I␣just␣computed␣is␣out␣of␣range.",
>   "So␣it␣will␣probably␣look␣funny.␣Proceed,␣for␣a␣laugh.", Λ};
>   *mp_back_error*(*mp*, "Some␣number␣got␣too␣big", *hlp*, *true*);
>   ;
>   *mp_get_x_next*(*mp*);
>   *mp*→*arith_error* = *false*;
> }

This code is used in section 328.

331.   Two knots in a row with the same coordinates will always be joined by an explicit "curve" whose control points are identical with the knots.

⟨ If consecutive knots are equal, join them explicitly 331 ⟩ ≡

```
p = knots; do
{
  q = mp_next_knot(p);
  if (number_equal(p→x_coord, q→x_coord) ∧ number_equal(p→y_coord,
          q→y_coord) ∧ mp_right_type(p) > mp_explicit) {
    mp_right_type(p) = mp_explicit;
    if (mp_left_type(p) ≡ mp_open) {
      mp_left_type(p) = mp_curl;
      set_number_to_unity(p→left_curl);
    }
    mp_left_type(q) = mp_explicit;
    if (mp_right_type(q) ≡ mp_open) {
      mp_right_type(q) = mp_curl;
      set_number_to_unity(q→right_curl);
    }
    number_clone(p→right_x, p→x_coord);
    number_clone(q→left_x, p→x_coord);
    number_clone(p→right_y, p→y_coord);
    number_clone(q→left_y, p→y_coord);
  }
  p = q;
}
while (p ≠ knots)
```

This code is used in section 328.

332.   If there are no breakpoints, it is necessary to compute the direction angles around an entire cycle. In this case the *mp_left_type* of the first node is temporarily changed to *end_cycle*.

⟨ Find the first breakpoint, h, on the path; insert an artificial breakpoint if the path is an unbroken
        cycle 332 ⟩ ≡

```
h = knots;
while (1) {
  if (mp_left_type(h) ≠ mp_open) break;
  if (mp_right_type(h) ≠ mp_open) break;
  h = mp_next_knot(h);
  if (h ≡ knots) {
    mp_left_type(h) = mp_end_cycle;
    break;
  }
}
```

This code is used in section 328.

333.   If $mp\_right\_type(p) < given$ and $q = mp\_link(p)$, we must have $mp\_right\_type(p) = mp\_left\_type(q) = mp\_explicit$ or $endpoint$.

$\langle$ Fill in the control points between p and the next breakpoint, then advance p to that breakpoint 333 $\rangle \equiv$
```
  q = mp_next_knot(p);
  if (mp_right_type(p) ≥ mp_given) {
     while ((mp_left_type(q) ≡ mp_open) ∧ (mp_right_type(q) ≡ mp_open)) {
       q = mp_next_knot(q);
     }
     ⟨ Fill in the control information between consecutive breakpoints p and q 339 ⟩;
  }
  else if (mp_right_type(p) ≡ mp_endpoint) {
     ⟨ Give reasonable values for the unused control points between p and q 334 ⟩;
  }
  p = q
```
This code is used in section 328.

334.   This step makes it possible to transform an explicitly computed path without checking the mp_left_type ▌ and mp_right_type fields.

$\langle$ Give reasonable values for the unused control points between p and q 334 $\rangle \equiv$
```
  {
     number_clone(p⃗right_x, p⃗x_coord);
     number_clone(p⃗right_y, p⃗y_coord);
     number_clone(q⃗left_x, q⃗x_coord);
     number_clone(q⃗left_y, q⃗y_coord);
  }
```
This code is used in section 333.

335.   Before we can go further into the way choices are made, we need to consider the underlying theory. The basic ideas implemented in *make_choices* are due to John Hobby, who introduced the notion of "mock curvature" at a knot. Angles are chosen so that they preserve mock curvature when a knot is passed, and this has been found to produce excellent results.

It is convenient to introduce some notations that simplify the necessary formulas. Let $d_{k,k+1} = |z_{k+1} - z_k|$ be the (nonzero) distance between knots k and $k + 1$; and let

$$\frac{z_{k+1} - z_k}{z_k - z_{k-1}} = \frac{d_{k,k+1}}{d_{k-1,k}} e^{i\psi_k}$$

so that a polygonal line from z_{k-1} to z_k to z_{k+1} turns left through an angle of ψ_k. We assume that $|\psi_k|Ł180°$. The control points for the spline from z_k to z_{k+1} will be denoted by

$$z_k^+ = z_k + \tfrac{1}{3}\rho_k e^{i\theta_k}(z_{k+1} - z_k),$$
$$z_{k+1}^- = z_{k+1} - \tfrac{1}{3}\sigma_{k+1}e^{-i\phi_{k+1}}(z_{k+1} - z_k),$$

where ρ_k and σ_{k+1} are nonnegative "velocity ratios" at the beginning and end of the curve, while θ_k and ϕ_{k+1} are the corresponding "offset angles." These angles satisfy the condition

$$\theta_k + \phi_k + \psi_k = 0, \tag{$*$}$$

whenever the curve leaves an intermediate knot k in the direction that it enters.

336.   Let α_k and β_{k+1} be the reciprocals of the "tension" of the curve at its beginning and ending points. This means that $\rho_k = \alpha_k f(\theta_k, \phi_{k+1})$ and $\sigma_{k+1} = \beta_{k+1} f(\phi_{k+1}, \theta_k)$, where $f(\theta, \phi)$ is METAPOST's standard velocity function defined in the *velocity* subroutine. The cubic spline $B(z_k, z_k^+, z_{k+1}^-, z_{k+1}; t)$ has curvature

$$\frac{2\sigma_{k+1}\sin(\theta_k + \phi_{k+1}) - 6\sin\theta_k}{\rho_k^2 d_{k,k+1}} \qquad \text{and} \qquad \frac{2\rho_k \sin(\theta_k + \phi_{k+1}) - 6\sin\phi_{k+1}}{\sigma_{k+1}^2 d_{k,k+1}}$$

at $t = 0$ and $t = 1$, respectively. The mock curvature is the linear approximation to this true curvature that arises in the limit for small θ_k and ϕ_{k+1}, if second-order terms are discarded. The standard velocity function satisfies

$$f(\theta, \phi) = 1 + O(\theta^2 + \theta\phi + \phi^2);$$

hence the mock curvatures are respectively

$$\frac{2\beta_{k+1}(\theta_k + \phi_{k+1}) - 6\theta_k}{\alpha_k^2 d_{k,k+1}} \qquad \text{and} \qquad \frac{2\alpha_k(\theta_k + \phi_{k+1}) - 6\phi_{k+1}}{\beta_{k+1}^2 d_{k,k+1}}. \tag{$**$}$$

337.   The turning angles ψ_k are given, and equation ($*$) above determines ϕ_k when θ_k is known, so the task of angle selection is essentially to choose appropriate values for each θ_k. When equation ($*$) is used to eliminate ϕ variables from ($**$), we obtain a system of linear equations of the form

$$A_k\theta_{k-1} + (B_k + C_k)\theta_k + D_k\theta_{k+1} = -B_k\psi_k - D_k\psi_{k+1},$$

where

$$A_k = \frac{\alpha_{k-1}}{\beta_k^2 d_{k-1,k}}, \qquad B_k = \frac{3 - \alpha_{k-1}}{\beta_k^2 d_{k-1,k}}, \qquad C_k = \frac{3 - \beta_{k+1}}{\alpha_k^2 d_{k,k+1}}, \qquad D_k = \frac{\beta_{k+1}}{\alpha_k^2 d_{k,k+1}}.$$

The tensions are always $\frac{3}{4}$ or more, hence each α and β will be at most $\frac{4}{3}$. It follows that $B_k \geq \frac{5}{4}A_k$ and $C_k \geq \frac{5}{4}D_k$; hence the equations are diagonally dominant; hence they have a unique solution. Moreover, in most cases the tensions are equal to 1, so that $B_k = 2A_k$ and $C_k = 2D_k$. This makes the solution numerically stable, and there is an exponential damping effect: The data at knot $k \pm j$ affects the angle at knot k by a factor of $O(2^{-j})$.

338.   However, we still must consider the angles at the starting and ending knots of a non-cyclic path. These angles might be given explicitly, or they might be specified implicitly in terms of an amount of "curl."

Let's assume that angles need to be determined for a non-cyclic path starting at z_0 and ending at z_n. Then equations of the form

$$A_k\theta_{k-1} + (B_k + C_k)\theta_k + D_k\theta_{k+1} = R_k$$

have been given for $0 < k < n$, and it will be convenient to introduce equations of the same form for $k = 0$ and $k = n$, where

$$A_0 = B_0 = C_n = D_n = 0.$$

If θ_0 is supposed to have a given value E_0, we simply define $C_0 = 1$, $D_0 = 0$, and $R_0 = E_0$. Otherwise a curl parameter, γ_0, has been specified at z_0; this means that the mock curvature at z_0 should be γ_0 times the mock curvature at z_1; i.e.,

$$\frac{2\beta_1(\theta_0 + \phi_1) - 6\theta_0}{\alpha_0^2 d_{01}} = \gamma_0 \frac{2\alpha_0(\theta_0 + \phi_1) - 6\phi_1}{\beta_1^2 d_{01}}.$$

This equation simplifies to

$$(\alpha_0\chi_0 + 3 - \beta_1)\theta_0 + \big((3 - \alpha_0)\chi_0 + \beta_1\big)\theta_1 = -\big((3 - \alpha_0)\chi_0 + \beta_1\big)\psi_1,$$

where $\chi_0 = \alpha_0^2\gamma_0/\beta_1^2$; so we can set $C_0 = \chi_0\alpha_0 + 3 - \beta_1$, $D_0 = (3 - \alpha_0)\chi_0 + \beta_1$, $R_0 = -D_0\psi_1$. It can be shown that $C_0 > 0$ and $C_0B_1 - A_1D_0 > 0$ when $\gamma_0 \geq 0$, hence the linear equations remain nonsingular.

Similar considerations apply at the right end, when the final angle ϕ_n may or may not need to be determined. It is convenient to let $\psi_n = 0$, hence $\theta_n = -\phi_n$. We either have an explicit equation $\theta_n = E_n$, or we have

$$\big((3 - \beta_n)\chi_n + \alpha_{n-1}\big)\theta_{n-1} + (\beta_n\chi_n + 3 - \alpha_{n-1})\theta_n = 0, \qquad \chi_n = \frac{\beta_n^2\gamma_n}{\alpha_{n-1}^2}.$$

When *make_choices* chooses angles, it must compute the coefficients of these linear equations, then solve the equations. To compute the coefficients, it is necessary to compute arctangents of the given turning angles ψ_k. When the equations are solved, the chosen directions θ_k are put back into the form of control points by essentially computing sines and cosines.

339.   OK, we are ready to make the hard choices of *make_choices*. Most of the work is relegated to an auxiliary procedure called *solve_choices*, which has been introduced to keep *make_choices* from being extremely long.

⟨Fill in the control information between consecutive breakpoints p and q 339⟩ ≡
  ⟨Calculate the turning angles ψ_k and the distances $d_{k,k+1}$; set n to the length of the path 343⟩;
  ⟨Remove *open* types at the breakpoints 344⟩;
  mp_solve_choices(*mp*, *p*, *q*, *n*)
This code is used in section 333.

340.   It's convenient to precompute quantities that will be needed several times later. The values of *delta_x*[k] and *delta_y*[k] will be the coordinates of $z_{k+1} - z_k$, and the magnitude of this vector will be *delta*[k] = $d_{k,k+1}$. The path angle ψ_k between $z_k - z_{k-1}$ and $z_{k+1} - z_k$ will be stored in *psi*[k].

⟨Global variables 14⟩ +≡
  int *path_size*;   /* maximum number of knots between breakpoints of a path */
  mp_number *delta_x*;
  mp_number *delta_y*;
  mp_number *delta*;   /* knot differences */
  mp_number *psi*;   /* turning angles */

341.   ⟨Dealloc variables 27⟩ +≡
```
  {
    int k;
    for (k = 0;  k < mp→path_size;  k++) {
      free_number (mp→delta_x[k]);
      free_number (mp→delta_y[k]);
      free_number (mp→delta[k]);
      free_number (mp→psi[k]);
    }
    xfree (mp→delta_x);
    xfree (mp→delta_y);
    xfree (mp→delta);
    xfree (mp→psi);
  }
```

342.   ⟨Other local variables for *make_choices* 342⟩ ≡
```
  int k, n;      /* current and final knot numbers */
  mp_knot s, t;    /* registers for list traversal */
```
This code is used in section 328.

343.   ⟨ Calculate the turning angles ψ_k and the distances $d_{k,k+1}$; set n to the length of the path 343 ⟩ ≡

```
{
   mp_number sine, cosine;      /* trig functions of various angles */
   new_fraction(sine);
   new_fraction(cosine);
RESTART:  k = 0;
   s = p;
   n = mp→path_size;
   do {
      t = mp_next_knot(s);
      set_number_from_substraction(mp→delta_x[k], t→x_coord, s→x_coord);
      set_number_from_substraction(mp→delta_y[k], t→y_coord, s→y_coord);
      pyth_add(mp→delta[k], mp→delta_x[k], mp→delta_y[k]);
      if (k > 0) {
         mp_number arg1, arg2, r1, r2;

         new_number(arg1);
         new_number(arg2);
         new_fraction(r1);
         new_fraction(r2);
         make_fraction(r1, mp→delta_y[k − 1], mp→delta[k − 1]);
         number_clone(sine, r1);
         make_fraction(r2, mp→delta_x[k − 1], mp→delta[k − 1]);
         number_clone(cosine, r2);
         take_fraction(r1, mp→delta_x[k], cosine);
         take_fraction(r2, mp→delta_y[k], sine);
         set_number_from_addition(arg1, r1, r2);
         take_fraction(r1, mp→delta_y[k], cosine);
         take_fraction(r2, mp→delta_x[k], sine);
         set_number_from_substraction(arg2, r1, r2);
         n_arg(mp→psi[k], arg1, arg2);
         free_number(r1);
         free_number(r2);
         free_number(arg1);
         free_number(arg2);
      }
      incr(k);
      s = t;
      if (k ≡ mp→path_size) {
         mp_reallocate_paths(mp, mp→path_size + (mp→path_size/4));
         goto RESTART;      /* retry, loop size has changed */
      }
      if (s ≡ q)  n = k;
   } while (¬((k ≥ n) ∧ (mp_left_type(s) ≠ mp_end_cycle)));
   if (k ≡ n)  set_number_to_zero(mp→psi[k]);
   else  number_clone(mp→psi[k], mp→psi[1]);
   free_number(sine);
   free_number(cosine);
}
```

This code is used in section 339.

344.   When we get to this point of the code, $mp\_right\_type(p)$ is either *given* or *curl* or *open*. If it is *open*, we must have $mp\_left\_type(p) = mp\_end\_cycle$ or $mp\_left\_type(p) = mp\_explicit$. In the latter case, the *open* type is converted to *given*; however, if the velocity coming into this knot is zero, the *open* type is converted to a *curl*, since we don't know the incoming direction.

Similarly, $mp\_left\_type(q)$ is either *given* or *curl* or *open* or mp_end_cycle. The *open* possibility is reduced either to *given* or to *curl*.

⟨ Remove *open* types at the breakpoints 344 ⟩ ≡
```
{
  mp_number delx, dely;        /* directions where open meets explicit */
  new_number(delx);
  new_number(dely);
  if (mp_left_type(q) ≡ mp_open) {
    set_number_from_substraction(delx, q→right_x, q→x_coord);
    set_number_from_substraction(dely, q→right_y, q→y_coord);
    if (number_zero(delx) ∧ number_zero(dely)) {
      mp_left_type(q) = mp_curl;
      set_number_to_unity(q→left_curl);
    }
    else {
      mp_left_type(q) = mp_given;
      n_arg(q→left_given, delx, dely);
    }
  }
  if ((mp_right_type(p) ≡ mp_open) ∧ (mp_left_type(p) ≡ mp_explicit)) {
    set_number_from_substraction(delx, p→x_coord, p→left_x);
    set_number_from_substraction(dely, p→y_coord, p→left_y);
    if (number_zero(delx) ∧ number_zero(dely)) {
      mp_right_type(p) = mp_curl;
      set_number_to_unity(p→right_curl);
    }
    else {
      mp_right_type(p) = mp_given;
      n_arg(p→right_given, delx, dely);
    }
  }
  free_number(delx);
  free_number(dely);
}
```
This code is used in section 339.

345.   Linear equations need to be solved whenever $n > 1$; and also when $n = 1$ and exactly one of the breakpoints involves a curl. The simplest case occurs when $n = 1$ and there is a curl at both breakpoints; then we simply draw a straight line.

But before coding up the simple cases, we might as well face the general case, since we must deal with it sooner or later, and since the general case is likely to give some insight into the way simple cases can be handled best.

When there is no cycle, the linear equations to be solved form a tridiagonal system, and we can apply the standard technique of Gaussian elimination to convert that system to a sequence of equations of the form

$$\theta_0 + u_0\theta_1 = v_0, \quad \theta_1 + u_1\theta_2 = v_1, \quad \ldots, \quad \theta_{n-1} + u_{n-1}\theta_n = v_{n-1}, \quad \theta_n = v_n.$$

It is possible to do this diagonalization while generating the equations. Once θ_n is known, it is easy to determine $\theta_{n-1}, \ldots, \theta_1, \theta_0$; thus, the equations will be solved.

The procedure is slightly more complex when there is a cycle, but the basic idea will be nearly the same. In the cyclic case the right-hand sides will be $v_k + w_k\theta_0$ instead of simply v_k, and we will start the process off with $u_0 = v_0 = 0$, $w_0 = 1$. The final equation will be not $\theta_n = v_n$ but $\theta_n + u_n\theta_1 = v_n + w_n\theta_0$; an appropriate ending routine will take account of the fact that $\theta_n = \theta_0$ and eliminate the w's from the system, after which the solution can be obtained as before.

When u_k, v_k, and w_k are being computed, the three pointer variables r, s, t will point respectively to knots $k - 1$, k, and $k + 1$. The u's and w's are scaled by 2^{28}, i.e., they are of type *fraction*; the θ's and v's are of type *angle*.

⟨ Global variables 14 ⟩ +≡
    mp_number *∗theta*;   /∗ values of θ_k ∗/
    mp_number *∗uu*;   /∗ values of u_k ∗/
    mp_number *∗vv*;   /∗ values of v_k ∗/
    mp_number *∗ww*;   /∗ values of w_k ∗/

346.   ⟨ Dealloc variables 27 ⟩ +≡
  {
    int *k*;
    for (*k* = 0; *k* < *mp→path_size*; *k*++) {
      free_number (*mp→theta*[*k*]);
      free_number (*mp→uu*[*k*]);
      free_number (*mp→vv*[*k*]);
      free_number (*mp→ww*[*k*]);
    }
    xfree (*mp→theta*);
    xfree (*mp→uu*);
    xfree (*mp→vv*);
    xfree (*mp→ww*);
  }

347.   ⟨ Declarations 8 ⟩ +≡
  static void *mp_reallocate_paths* (**MP** *mp*, **int** *l*);

348.   **void** $mp\_reallocate\_paths(\textbf{MP}\ mp, \textbf{int}\ l)$

$\{$

　　int k;

　　XREALLOC($mp \rightarrow delta\_x, l, \textbf{mp\_number}$);
　　XREALLOC($mp \rightarrow delta\_y, l, \textbf{mp\_number}$);
　　XREALLOC($mp \rightarrow delta, l, \textbf{mp\_number}$);
　　XREALLOC($mp \rightarrow psi, l, \textbf{mp\_number}$);
　　XREALLOC($mp \rightarrow theta, l, \textbf{mp\_number}$);
　　XREALLOC($mp \rightarrow uu, l, \textbf{mp\_number}$);
　　XREALLOC($mp \rightarrow vv, l, \textbf{mp\_number}$);
　　XREALLOC($mp \rightarrow ww, l, \textbf{mp\_number}$);
　　for $(k = mp \rightarrow path\_size;\ k < l;\ k{+}{+})\ \{$
　　　　$new\_number(mp \rightarrow delta\_x[k])$;
　　　　$new\_number(mp \rightarrow delta\_y[k])$;
　　　　$new\_number(mp \rightarrow delta[k])$;
　　　　$new\_angle(mp \rightarrow psi[k])$;
　　　　$new\_angle(mp \rightarrow theta[k])$;
　　　　$new\_fraction(mp \rightarrow uu[k])$;
　　　　$new\_angle(mp \rightarrow vv[k])$;
　　　　$new\_fraction(mp \rightarrow ww[k])$;
　　$\}$
　　$mp \rightarrow path\_size = l$;
$\}$

349.   Our immediate problem is to get the ball rolling by setting up the first equation or by realizing that no equations are needed, and to fit this initialization into a framework suitable for the overall computation.

⟨ Declarations 8 ⟩ +≡

　　static void $mp\_solve\_choices(\textbf{MP}\ mp, \textbf{mp\_knot}\ p, \textbf{mp\_knot}\ q, \textbf{halfword}\ n)$;

350.   **void** $mp\_solve\_choices$(**MP** mp, **mp_knot** p, **mp_knot** q, **halfword** n)
{
   int k;   /∗ current knot number ∗/
   mp_knot r, s, t;   /∗ registers for list traversal ∗/
   mp_number ff;
   $new\_fraction(ff)$;
   `FUNCTION_TRACE2("solve_choices(%d)\n"`, n);
   $k = 0$;
   $s = p$;
   $r = 0$;
   while (1) {
     $t = mp\_next\_knot(s)$;
     if ($k \equiv 0$) {⟨ Get the linear equations started; or **return** with the control points in place, if linear
        equations needn't be solved 351 ⟩}
     else {
       switch ($mp\_left\_type(s)$) {
       case mp_end_cycle: **case** mp_open: ⟨ Set up equation to match mock curvatures at z_k; then **goto**
          $found$ with θ_n adjusted to equal θ_0, if a cycle has ended 353 ⟩;
        break;
       case mp_curl: ⟨ Set up equation for a curl at θ_n and **goto** $found$ 363 ⟩;
        break;
       case mp_given: ⟨ Calculate the given value of θ_n and **goto** $found$ 360 ⟩;
        break;
       }   /∗ there are no other cases ∗/
     }
     $r = s$;
     $s = t$;
     $incr(k)$;
   }
 `FOUND`: ⟨ Finish choosing angles and assigning control points 366 ⟩;
   $free\_number(ff)$;
}

351.   On the first time through the loop, we have $k = 0$ and r is not yet defined. The first linear equation, if any, will have $A_0 = B_0 = 0$.

⟨ Get the linear equations started; or **return** with the control points in place, if linear equations needn't be
        solved 351 ⟩ ≡

```
switch (mp_right_type(s)) {
case mp_given:
    if (mp_left_type(t) ≡ mp_given) {⟨Reduce to simple case of two givens and return 373⟩}
    else {
        ⟨ Set up the equation for a given value of θ₀ 361 ⟩;
    }
    break;
case mp_curl:
    if (mp_left_type(t) ≡ mp_curl) {⟨Reduce to simple case of straight line and return 374⟩}
    else {
        ⟨ Set up the equation for a curl at θ₀ 362 ⟩;
    }
    break;
case mp_open: set_number_to_zero(mp→uu[0]);
    set_number_to_zero(mp→vv[0]);
    number_clone(mp→ww[0], fraction_one_t);      /* this begins a cycle */
    break;
}       /* there are no other cases */
```
This code is used in section 350.

352.   The general equation that specifies equality of mock curvature at z_k is

$$A_k\theta_{k-1} + (B_k + C_k)\theta_k + D_k\theta_{k+1} = -B_k\psi_k - D_k\psi_{k+1},$$

as derived above. We want to combine this with the already-derived equation $\theta_{k-1}+u_{k-1}\theta_k = v_{k-1}+w_{k-1}\theta_0$ in order to obtain a new equation $\theta_k + u_k\theta_{k+1} = v_k + w_k\theta_0$. This can be done by dividing the equation

$$(B_k - u_{k-1}A_k + C_k)\theta_k + D_k\theta_{k+1} = -B_k\psi_k - D_k\psi_{k+1} - A_kv_{k-1} - A_kw_{k-1}\theta_0$$

by $B_k - u_{k-1}A_k + C_k$. The trick is to do this carefully with fixed-point arithmetic, avoiding the chance of overflow while retaining suitable precision.

   The calculations will be performed in several registers that provide temporary storage for intermediate quantities.

353.   ⟨Set up equation to match mock curvatures at z_k; then **goto** *found* with θ_n adjusted to equal θ_0, if a cycle has ended 353⟩ ≡

```
{
  mp_number aa, bb, cc, acc;     /* temporary registers */
  mp_number dd, ee;       /* likewise, but scaled */
  new_fraction(aa);
  new_fraction(bb);
  new_fraction(cc);
  new_fraction(acc);
  new_number(dd);
  new_number(ee);
```
⟨Calculate the values $aa = A_k/B_k$, $bb = D_k/C_k$, $dd = (3 - \alpha_{k-1})d_{k,k+1}$, $ee = (3 - \beta_{k+1})d_{k-1,k}$, and $cc = (B_k - u_{k-1}A_k)/B_k$ 354⟩;
⟨Calculate the ratio $ff = C_k/(C_k + B_k - u_{k-1}A_k)$ 355⟩;
take_fraction($mp{\rightarrow}uu[k]$, ff, bb);
⟨Calculate the values of v_k and w_k 356⟩;
```
  if (mp_left_type(s) ≡ mp_end_cycle) {
```
⟨Adjust θ_n to equal θ_0 and **goto** *found* 357⟩;
```
  }
  free_number(aa);
  free_number(bb);
  free_number(cc);
  free_number(acc);
  free_number(dd);
  free_number(ee);
}
```
This code is used in section 350.

354.   Since tension values are never less than $3/4$, the values aa and bb computed here are never more than $4/5$.

$\langle$ Calculate the values $aa = A_k/B_k$, $bb = D_k/C_k$, $dd = (3 - \alpha_{k-1})d_{k,k+1}$, $ee = (3 - \beta_{k+1})d_{k-1,k}$, and
$\quad cc = (B_k - u_{k-1}A_k)/B_k$ 354 $\rangle \equiv$
{
    mp_number $absval$;

    $new\_number(absval)$;
    $number\_clone(absval, r\text{-}right\_tension)$;
    $number\_abs(absval)$;
    if $(number\_equal(absval, unity\_t))$ {
        $number\_clone(aa, fraction\_half\_t)$;
        $number\_clone(dd, mp\text{-}delta[k])$;
        $number\_double(dd)$;
    }
    else {
        mp_number $arg1$, $arg2$, ret;

        $new\_number(arg2)$;
        $new\_number(arg1)$;
        $number\_clone(arg2, r\text{-}right\_tension)$;
        $number\_abs(arg2)$;
        $number\_multiply\_int(arg2, 3)$;
        $number\_substract(arg2, unity\_t)$;
        $make\_fraction(aa, unity\_t, arg2)$;
        $number\_clone(arg2, r\text{-}right\_tension)$;
        $number\_abs(arg2)$;
        $new\_fraction(ret)$;
        $make\_fraction(ret, unity\_t, arg2)$;
        $set\_number\_from\_substraction(arg1, fraction\_three\_t, ret)$;
        $take\_fraction(arg2, mp\text{-}delta[k], arg1)$;
        $number\_clone(dd, arg2)$;
        $free\_number(ret)$;
        $free\_number(arg1)$;
        $free\_number(arg2)$;
    }
    $number\_clone(absval, t\text{-}left\_tension)$;
    $number\_abs(absval)$;
    if $(number\_equal(absval, unity\_t))$ {
        $number\_clone(bb, fraction\_half\_t)$;
        $number\_clone(ee, mp\text{-}delta[k - 1])$;
        $number\_double(ee)$;
    }
    else {
        mp_number $arg1$, $arg2$, ret;

        $new\_number(arg1)$;
        $new\_number(arg2)$;
        $number\_clone(arg2, t\text{-}left\_tension)$;
        $number\_abs(arg2)$;
        $number\_multiply\_int(arg2, 3)$;
        $number\_substract(arg2, unity\_t)$;
        $make\_fraction(bb, unity\_t, arg2)$;
        $number\_clone(arg2, t\text{-}left\_tension)$;

$number\_abs(arg2)$;
$new\_fraction(ret)$;
$make\_fraction(ret, unity\_t, arg2)$;
$set\_number\_from\_substraction(arg1, fraction\_three\_t, ret)$;
$take\_fraction(ee, mp \rightarrow delta[k-1], arg1)$;
$free\_number(ret)$;
$free\_number(arg1)$;
$free\_number(arg2)$;
}
$free\_number(absval)$;
}
{
mp_number $r1$;

$new\_number(r1)$;
$take\_fraction(r1, mp \rightarrow uu[k-1], aa)$;
$set\_number\_from\_substraction(cc, fraction\_one\_t, r1)$;
$free\_number(r1)$;
}
This code is used in section 353.

355.   The ratio to be calculated in this step can be written in the form

$$\frac{\beta_k^2 \cdot ee}{\beta_k^2 \cdot ee + \alpha_k^2 \cdot cc \cdot dd,}$$

because of the quantities just calculated. The values of dd and ee will not be needed after this step has been performed.

⟨ Calculate the ratio $ff = C_k/(C_k + B_k - u_{k-1}A_k)$ 355 ⟩ ≡
```
{
    mp_number rt, lt;
    mp_number arg2;

    new_number(arg2);
    number_clone(arg2, dd);
    take_fraction(dd, arg2, cc);
    new_number(lt);
    new_number(rt);
    number_clone(lt, s→left_tension);
    number_abs(lt);
    number_clone(rt, s→right_tension);
    number_abs(rt);
    if (¬number_equal(lt, rt)) {      /* β_k^{-1} ≠ α_k^{-1} */
        mp_number r1;

        new_number(r1);
        if (number_less(lt, rt)) {
            make_fraction(r1, lt, rt);      /* α_k^2/β_k^2 */
            take_fraction(ff, r1, r1);
            number_clone(r1, dd);
            take_fraction(dd, r1, ff);
        }
        else {
            make_fraction(r1, rt, lt);      /* β_k^2/α_k^2 */
            take_fraction(ff, r1, r1);
            number_clone(r1, ee);
            take_fraction(ee, r1, ff);
        }
        free_number(r1);
    }
    free_number(rt);
    free_number(lt);
    set_number_from_addition(arg2, dd, ee);
    make_fraction(ff, ee, arg2);
    free_number(arg2);
}
```
This code is used in section 353.

356.   The value of u_{k-1} will be ≤ 1 except when $k = 1$ and the previous equation was specified by a curl. In that case we must use a special method of computation to prevent overflow.

Fortunately, the calculations turn out to be even simpler in this "hard" case. The curl equation makes $w_0 = 0$ and $v_0 = -u_0\psi_1$, hence $-B_1\psi_1 - A_1v_0 = -(B_1 - u_0A_1)\psi_1 = -cc \cdot B_1\psi_1$.

⟨ Calculate the values of v_k and w_k 356 ⟩ ≡
  take_fraction(*acc*, *mp→psi*[*k* + 1], *mp→uu*[*k*]);
  number_negate(*acc*);
  if (*mp_right_type*(*r*) ≡ *mp_curl*) {
    mp_number *r1*, *arg2*;

    new_fraction(*r1*);
    new_number(*arg2*);
    set_number_from_substraction(*arg2*, *fraction_one_t*, *ff*);
    take_fraction(*r1*, *mp→psi*[1], *arg2*);
    set_number_to_zero(*mp→ww*[*k*]);
    set_number_from_substraction(*mp→vv*[*k*], *acc*, *r1*);
    free_number(*r1*);
    free_number(*arg2*);
  }
  else {
    mp_number *arg1*, *r1*;

    new_fraction(*r1*);
    new_number(*arg1*);
    set_number_from_substraction(*arg1*, *fraction_one_t*, *ff*);
    make_fraction(*ff*, *arg1*, *cc*);   /* this is $B_k/(C_k + B_k - u_{k-1}A_k) < 5$ */
    free_number(*arg1*);
    take_fraction(*r1*, *mp→psi*[*k*], *ff*);
    number_substract(*acc*, *r1*);
    number_clone(*r1*, *ff*);
    take_fraction(*ff*, *r1*, *aa*);   /* this is $A_k/(C_k + B_k - u_{k-1}A_k)$ */
    take_fraction(*r1*, *mp→vv*[*k* − 1], *ff*);
    set_number_from_substraction(*mp→vv*[*k*], *acc*, *r1*);
    if (*number_zero*(*mp→ww*[*k* − 1])) {
      set_number_to_zero(*mp→ww*[*k*]);
    }
    else {
      take_fraction(*mp→ww*[*k*], *mp→ww*[*k* − 1], *ff*);
      number_negate(*mp→ww*[*k*]);
    }
    free_number(*r1*);
  }

This code is used in section 353.

357.   When a complete cycle has been traversed, we have $\theta_k + u_k\theta_{k+1} = v_k + w_k\theta_0$, for $1 \leq k \leq n$. We would like to determine the value of θ_n and reduce the system to the form $\theta_k + u_k\theta_{k+1} = v_k$ for $0 \leq k < n$, so that the cyclic case can be finished up just as if there were no cycle.

The idea in the following code is to observe that

$$\theta_n = v_n + w_n\theta_0 - u_n\theta_1 = \cdots$$
$$= v_n + w_n\theta_0 - u_n\big(v_1 + w_1\theta_0 - u_1(v_2 + \cdots - u_{n-2}(v_{n-1} + w_{n-1}\theta_0 - u_{n-1}\theta_0))\big),$$

so we can solve for $\theta_n = \theta_0$.

⟨ Adjust θ_n to equal θ_0 and **goto** *found* 357 ⟩ ≡

```
{
  mp_number arg2, r1;
  new_number(arg2);
  new_number(r1);
  set_number_to_zero(aa);
  number_clone(bb, fraction_one_t);      /* we have k = n */
  do {
    decr(k);
    if (k ≡ 0)  k = n;
    take_fraction(r1, aa, mp→uu[k]);
    set_number_from_substraction(aa, mp→vv[k], r1);
    take_fraction(r1, bb, mp→uu[k]);
    set_number_from_substraction(bb, mp→ww[k], r1);
  } while (k ≠ n);      /* now θₙ = aa + bb · θₙ */
  set_number_from_substraction(arg2, fraction_one_t, bb);
  make_fraction(r1, aa, arg2);
  number_clone(aa, r1);
  number_clone(mp→theta[n], aa);
  number_clone(mp→vv[0], aa);
  for (k = 1; k < n; k++) {
    take_fraction(r1, aa, mp→ww[k]);
    number_add(mp→vv[k], r1);
  }
  free_number(arg2);
  free_number(r1);
  free_number(aa);
  free_number(bb);
  free_number(cc);
  free_number(acc);
  free_number(dd);
  free_number(ee);
  goto FOUND;
}
```

This code is used in section 353.

358.   **void** $mp\_reduce\_angle(\textbf{MP}\ mp, \textbf{mp\_number}\ *a)$
  {
    mp_number abs_a;
    FUNCTION_TRACE2("reduce_angle(%f)\n", $number\_to\_double(*a)$);
    $new\_number(abs\_a)$;
    $number\_clone(abs\_a, *a)$;
    $number\_abs(abs\_a)$;
    if $(number\_greater(abs\_a, one\_eighty\_deg\_t))$ {
      if $(number\_positive(*a))$ {
        $number\_substract(*a, three\_sixty\_deg\_t)$;
      }
      else {
        $number\_add(*a, three\_sixty\_deg\_t)$;
      }
    }
    $free\_number(abs\_a)$;
  }

359.   ⟨ Declarations 8 ⟩ +≡
  void $mp\_reduce\_angle(\textbf{MP}\ mp, \textbf{mp\_number}\ *a)$;

360.   ⟨ Calculate the given value of θ_n and **goto** *found* 360 ⟩ ≡
  {
    mp_number $narg$;

    $new\_angle(narg)$;
    $n\_arg(narg, mp\rightarrow delta\_x[n-1], mp\rightarrow delta\_y[n-1])$;
    $set\_number\_from\_substraction(mp\rightarrow theta[n], s\rightarrow left\_given, narg)$;
    $free\_number(narg)$;
    $mp\_reduce\_angle(mp, \&mp\rightarrow theta[n])$;
    goto FOUND;
  }
This code is used in section 350.

361.   ⟨ Set up the equation for a given value of θ_0 361 ⟩ ≡
  {
    mp_number $narg$;

    $new\_angle(narg)$;
    $n\_arg(narg, mp\rightarrow delta\_x[0], mp\rightarrow delta\_y[0])$;
    $set\_number\_from\_substraction(mp\rightarrow vv[0], s\rightarrow right\_given, narg)$;
    $free\_number(narg)$;
    $mp\_reduce\_angle(mp, \&mp\rightarrow vv[0])$;
    $set\_number\_to\_zero(mp\rightarrow uu[0])$;
    $set\_number\_to\_zero(mp\rightarrow ww[0])$;
  }
This code is used in section 351.

362.   ⟨Set up the equation for a curl at θ_0 362⟩ ≡

  {

    mp_number *lt*, *rt*, *cc*;   /∗ tension values ∗/

    new_number(*lt*);
    new_number(*rt*);
    new_number(*cc*);
    number_clone(*cc*, *s*→*right_curl*);
    number_clone(*lt*, *t*→*left_tension*);
    number_abs(*lt*);
    number_clone(*rt*, *s*→*right_tension*);
    number_abs(*rt*);
    if (*number_unity*(*rt*) ∧ *number_unity*(*lt*)) {
      mp_number *arg1*, *arg2*;

      new_number(*arg1*);
      new_number(*arg2*);
      number_clone(*arg1*, *cc*);
      number_double(*arg1*);
      number_add(*arg1*, *unity_t*);
      number_clone(*arg2*, *cc*);
      number_add(*arg2*, *two_t*);
      make_fraction(*mp*→*uu*[0], *arg1*, *arg2*);
      free_number(*arg1*);
      free_number(*arg2*);
    }
    else {
      mp_curl_ratio(*mp*, &*mp*→*uu*[0], *cc*, *rt*, *lt*);
    }
    take_fraction(*mp*→*vv*[0], *mp*→*psi*[1], *mp*→*uu*[0]);
    number_negate(*mp*→*vv*[0]);
    set_number_to_zero(*mp*→*ww*[0]);
    free_number(*rt*);
    free_number(*lt*);
    free_number(*cc*);
  }

This code is used in section 351.

363.   ⟨Set up equation for a curl at θ_n and **goto** *found* 363⟩ ≡
  {
    mp_number *lt*, *rt*, *cc*;   /∗ tension values ∗/
    new_number(*lt*);
    new_number(*rt*);
    new_number(*cc*);
    number_clone(*cc*, *s→left_curl*);
    number_clone(*lt*, *s→left_tension*);
    number_abs(*lt*);
    number_clone(*rt*, *r→right_tension*);
    number_abs(*rt*);
    if (*number_unity*(*rt*) ∧ *number_unity*(*lt*)) {
      mp_number *arg1*, *arg2*;

      new_number(*arg1*);
      new_number(*arg2*);
      number_clone(*arg1*, *cc*);
      number_double(*arg1*);
      number_add(*arg1*, *unity_t*);
      number_clone(*arg2*, *cc*);
      number_add(*arg2*, *two_t*);
      make_fraction(*ff*, *arg1*, *arg2*);
      free_number(*arg1*);
      free_number(*arg2*);
    }
    else {
      mp_curl_ratio(*mp*, &*ff*, *cc*, *lt*, *rt*);
    }
    {
      mp_number *arg1*, *arg2*, *r1*;

      new_fraction(*r1*);
      new_fraction(*arg1*);
      new_number(*arg2*);
      take_fraction(*arg1*, *mp→vv*[*n* − 1], *ff*);
      take_fraction(*r1*, *ff*, *mp→uu*[*n* − 1]);
      set_number_from_substraction(*arg2*, *fraction_one_t*, *r1*);
      make_fraction(*mp→theta*[*n*], *arg1*, *arg2*);
      number_negate(*mp→theta*[*n*]);
      free_number(*r1*);
      free_number(*arg1*);
      free_number(*arg2*);
    }
    free_number(*rt*);
    free_number(*lt*);
    free_number(*cc*);
    goto FOUND;
  }
This code is used in section 350.

364.   The *curl_ratio* subroutine has three arguments, which our previous notation encourages us to call γ, α^{-1}, and β^{-1}. It is a somewhat tedious program to calculate

$$\frac{(3-\alpha)\alpha^2\gamma + \beta^3}{\alpha^3\gamma + (3-\beta)\beta^2},$$

with the result reduced to 4 if it exceeds 4. (This reduction of curl is necessary only if the curl and tension are both large.) The values of α and β will be at most 4/3.

⟨ Declarations 8 ⟩ +≡
  static void *mp_curl_ratio*(**MP** *mp*, **mp_number** *∗ret*, **mp_number** *gamma*, **mp_number**
      a_tension, **mp_number** *b_tension*);

365.   **void** $mp\_curl\_ratio(\textbf{MP}\ mp, \textbf{mp\_number}\ *ret, \textbf{mp\_number}\ gamma\_orig, \textbf{mp\_number}$
$\quad\quad a\_tension, \textbf{mp\_number}\ b\_tension)$
{
  mp_number $alpha, beta, gamma, num, denom, ff$;   /∗ registers ∗/
  mp_number $arg1$;
  $new\_number(arg1)$;
  $new\_fraction(alpha)$;
  $new\_fraction(beta)$;
  $new\_fraction(gamma)$;
  $new\_fraction(ff)$;
  $new\_fraction(denom)$;
  $new\_fraction(num)$;
  $make\_fraction(alpha, unity\_t, a\_tension)$;
  $make\_fraction(beta, unity\_t, b\_tension)$;
  $number\_clone(gamma, gamma\_orig)$;
  if $(number\_lessequal(alpha, beta))$ {
    $make\_fraction(ff, alpha, beta)$;
    $number\_clone(arg1, ff)$;
    $take\_fraction(ff, arg1, arg1)$;
    $number\_clone(arg1, gamma)$;
    $take\_fraction(gamma, arg1, ff)$;
    $convert\_fraction\_to\_scaled(beta)$;
    $take\_fraction(denom, gamma, alpha)$;
    $number\_add(denom, three\_t)$;
  }
  else {
    $make\_fraction(ff, beta, alpha)$;
    $number\_clone(arg1, ff)$;
    $take\_fraction(ff, arg1, arg1)$;
    $take\_fraction(arg1, beta, ff)$;
    $convert\_fraction\_to\_scaled(arg1)$;
    $number\_clone(beta, arg1)$;
    $take\_fraction(denom, gamma, alpha)$;
    $set\_number\_from\_div(arg1, ff, twelvebits\_3)$;
    $number\_add(denom, arg1)$;
  }
  $number\_substract(denom, beta)$;
  $set\_number\_from\_substraction(arg1, fraction\_three\_t, alpha)$;
  $take\_fraction(num, gamma, arg1)$;
  $number\_add(num, beta)$;
  $number\_clone(arg1, denom)$;
  $number\_double(arg1)$;
  $number\_double(arg1)$;   /∗ arg1 = 4*denom ∗/
  if $(number\_greaterequal(num, arg1))$ {
    $number\_clone(*ret, fraction\_four\_t)$;
  }
  else {
    $make\_fraction(*ret, num, denom)$;
  }
  $free\_number(alpha)$;
  $free\_number(beta)$;
  $free\_number(gamma)$;

```
    free_number (num);
    free_number (denom);
    free_number (ff );
    free_number (arg1 );
  }
```

366.   We're in the home stretch now.

⟨ Finish choosing angles and assigning control points 366 ⟩ ≡
```
  {
    mp_number  r1 ;

    new_number (r1 );
    for (k = n − 1;  k ≥ 0;  k−−)  {
      take_fraction (r1 , mp→theta[k + 1], mp→uu[k]);
      set_number_from_substraction (mp→theta[k], mp→vv[k], r1 );
    }
    free_number (r1 );
  }
  s = p;
  k = 0;
  {
    mp_number  arg;

    new_number (arg);
    do  {
      t = mp_next_knot (s);
      n_sin_cos (mp→theta[k], mp→ct , mp→st );
      number_clone (arg, mp→psi [k + 1]);
      number_negate (arg);
      number_substract (arg, mp→theta[k + 1]);
      n_sin_cos (arg, mp→cf , mp→sf );
      mp_set_controls (mp, s, t, k);
      incr (k);
      s = t;
    } while (k ≠ n);
    free_number (arg);
  }
```
This code is used in section 350.

367.   The *set_controls* routine actually puts the control points into a pair of consecutive nodes p and q. Global variables are used to record the values of $\sin\theta$, $\cos\theta$, $\sin\phi$, and $\cos\phi$ needed in this calculation.

⟨ Global variables 14 ⟩ +≡
```
  mp_number  st ;
  mp_number  ct ;
  mp_number  sf ;
  mp_number  cf ;      /* sines and cosines */
```

368.   ⟨ Initialize table entries 182 ⟩ +≡
```
  new_fraction (mp→st );
  new_fraction (mp→ct );
  new_fraction (mp→sf );
  new_fraction (mp→cf );
```

369. ⟨Dealloc variables 27⟩ +≡

$free\_number(mp\rightarrow st)$;
$free\_number(mp\rightarrow ct)$;
$free\_number(mp\rightarrow sf)$;
$free\_number(mp\rightarrow cf)$;

370. ⟨Declarations 8⟩ +≡

static void $mp\_set\_controls(\textbf{MP}\ mp, \textbf{mp\_knot}\ p, \textbf{mp\_knot}\ q, \textbf{integer}\ k)$;

371.   **void** *mp_set_controls*(**MP** *mp*, **mp_knot** *p*, **mp_knot** *q*, **integer** *k*)
   {
   mp_number *rr*, *ss*;   /∗ velocities, divided by thrice the tension ∗/
   mp_number *lt*, *rt*;   /∗ tensions ∗/
   mp_number *sine*;   /∗ sin(θ + φ) ∗/
   mp_number *tmp*;
   mp_number *r1*, *r2*;

   new_number(*tmp*);
   new_number(*lt*);
   new_number(*rt*);
   new_number(*r1*);
   new_number(*r2*);
   number_clone(*lt*, *q⃗left_tension*);
   number_abs(*lt*);
   number_clone(*rt*, *p⃗right_tension*);
   number_abs(*rt*);
   new_fraction(*sine*);
   new_fraction(*rr*);
   new_fraction(*ss*);
   velocity(*rr*, *mp⃗st*, *mp⃗ct*, *mp⃗sf*, *mp⃗cf*, *rt*);
   velocity(*ss*, *mp⃗sf*, *mp⃗cf*, *mp⃗st*, *mp⃗ct*, *lt*);
   if (*number_negative*(*p⃗right_tension*) ∨ *number_negative*(*q⃗left_tension*)) {
      ⟨ Decrease the velocities, if necessary, to stay inside the bounding triangle 372 ⟩;
   }
   take_fraction(*r1*, *mp⃗delta_x*[*k*], *mp⃗ct*);
   take_fraction(*r2*, *mp⃗delta_y*[*k*], *mp⃗st*);
   number_substract(*r1*, *r2*);
   take_fraction(*tmp*, *r1*, *rr*);
   set_number_from_addition(*p⃗right_x*, *p⃗x_coord*, *tmp*);
   take_fraction(*r1*, *mp⃗delta_y*[*k*], *mp⃗ct*);
   take_fraction(*r2*, *mp⃗delta_x*[*k*], *mp⃗st*);
   number_add(*r1*, *r2*);
   take_fraction(*tmp*, *r1*, *rr*);
   set_number_from_addition(*p⃗right_y*, *p⃗y_coord*, *tmp*);
   take_fraction(*r1*, *mp⃗delta_x*[*k*], *mp⃗cf*);
   take_fraction(*r2*, *mp⃗delta_y*[*k*], *mp⃗sf*);
   number_add(*r1*, *r2*);
   take_fraction(*tmp*, *r1*, *ss*);
   set_number_from_substraction(*q⃗left_x*, *q⃗x_coord*, *tmp*);
   take_fraction(*r1*, *mp⃗delta_y*[*k*], *mp⃗cf*);
   take_fraction(*r2*, *mp⃗delta_x*[*k*], *mp⃗sf*);
   number_substract(*r1*, *r2*);
   take_fraction(*tmp*, *r1*, *ss*);
   set_number_from_substraction(*q⃗left_y*, *q⃗y_coord*, *tmp*);
   mp_right_type(*p*) = *mp_explicit*;
   mp_left_type(*q*) = *mp_explicit*;
   free_number(*tmp*);
   free_number(*r1*);
   free_number(*r2*);
   free_number(*lt*);
   free_number(*rt*);
   free_number(*rr*);

$\quad$ *free_number*(*ss*);
$\quad$ *free_number*(*sine*);
$\}$

372.   The boundedness conditions $rr\mathrm{L}\sin\phi\,/\sin(\theta+\phi)$ and $ss\mathrm{L}\sin\theta\,/\sin(\theta+\phi)$ are to be enforced if $\sin\theta$, $\sin\phi$, and $\sin(\theta+\phi)$ all have the same sign. Otherwise there is no "bounding triangle."

⟨ Decrease the velocities, if necessary, to stay inside the bounding triangle 372 ⟩ ≡
```
  if ((number_nonnegative(mp→st) ∧ number_nonnegative(mp→sf)) ∨ (number_nonpositive(mp→st) ∧
        number_nonpositive(mp→sf))) {
    mp_number r1, r2, arg1;
    mp_number ab_vs_cd;

    new_number(ab_vs_cd);
    new_fraction(r1);
    new_fraction(r2);
    new_number(arg1);
    number_clone(arg1, mp→st);
    number_abs(arg1);
    take_fraction(r1, arg1, mp→cf);
    number_clone(arg1, mp→sf);
    number_abs(arg1);
    take_fraction(r2, arg1, mp→ct);
    set_number_from_addition(sine, r1, r2);
    if (number_positive(sine)) {
      set_number_from_addition(arg1, fraction_one_t, unity_t);      /∗ safety factor ∗/
      number_clone(r1, sine);
      take_fraction(sine, r1, arg1);
      if (number_negative(p→right_tension)) {
        number_clone(arg1, mp→sf);
        number_abs(arg1);
        ab_vs_cd(ab_vs_cd, arg1, fraction_one_t, rr, sine);
        if (number_negative(ab_vs_cd)) {
          number_clone(arg1, mp→sf);
          number_abs(arg1);
          make_fraction(rr, arg1, sine);
        }
      }
      if (number_negative(q→left_tension)) {
        number_clone(arg1, mp→st);
        number_abs(arg1);
        ab_vs_cd(ab_vs_cd, arg1, fraction_one_t, ss, sine);
        if (number_negative(ab_vs_cd)) {
          number_clone(arg1, mp→st);
          number_abs(arg1);
          make_fraction(ss, arg1, sine);
        }
      }
    }
    free_number(arg1);
    free_number(r1);
    free_number(r2);
    free_number(ab_vs_cd);
  }
```
This code is used in section 371.

373.   Only the simple cases remain to be handled.

⟨ Reduce to simple case of two givens and **return** 373 ⟩ ≡
  {
    mp_number *arg1*;
    mp_number *narg*;

    new_angle(*narg*);
    n_arg(*narg*, *mp*→*delta_x*[0], *mp*→*delta_y*[0]);
    new_number(*arg1*);
    set_number_from_substraction(*arg1*, *p*→*right_given*, *narg*);
    n_sin_cos(*arg1*, *mp*→*ct*, *mp*→*st*);
    set_number_from_substraction(*arg1*, *q*→*left_given*, *narg*);
    n_sin_cos(*arg1*, *mp*→*cf*, *mp*→*sf*);
    number_negate(*mp*→*sf*);
    mp_set_controls(*mp*, *p*, *q*, 0);
    free_number(*narg*);
    free_number(*arg1*);
    free_number(*ff*);
    return;
  }
This code is used in section 351.

374.   ⟨Reduce to simple case of straight line and **return** 374⟩ ≡
  {
    mp_number *lt*, *rt*;   /* tension values */
    mp_right_type(*p*) = *mp_explicit*;
    mp_left_type(*q*) = *mp_explicit*;
    new_number(*lt*);
    new_number(*rt*);
    number_clone(*lt*, *q*→*left_tension*);
    number_abs(*lt*);
    number_clone(*rt*, *p*→*right_tension*);
    number_abs(*rt*);
    if (*number_unity*(*rt*)) {
      mp_number *arg2*;

      new_number(*arg2*);
      if (*number_nonnegative*(*mp*→*delta_x*[0])) {
        set_number_from_addition(*arg2*, *mp*→*delta_x*[0], *epsilon_t*);
      }
      else {
        set_number_from_substraction(*arg2*, *mp*→*delta_x*[0], *epsilon_t*);
      }
      number_int_div(*arg2*, 3);
      set_number_from_addition(*p*→*right_x*, *p*→*x_coord*, *arg2*);
      if (*number_nonnegative*(*mp*→*delta_y*[0])) {
        set_number_from_addition(*arg2*, *mp*→*delta_y*[0], *epsilon_t*);
      }
      else {
        set_number_from_substraction(*arg2*, *mp*→*delta_y*[0], *epsilon_t*);
      }
      number_int_div(*arg2*, 3);
      set_number_from_addition(*p*→*right_y*, *p*→*y_coord*, *arg2*);
      free_number(*arg2*);
    }
    else {
      mp_number *arg2*, *r1*;

      new_fraction(*r1*);
      new_number(*arg2*);
      number_clone(*arg2*, *rt*);
      number_multiply_int(*arg2*, 3);
      make_fraction(*ff*, *unity_t*, *arg2*);   /* α/3 */
      free_number(*arg2*);
      take_fraction(*r1*, *mp*→*delta_x*[0], *ff*);
      set_number_from_addition(*p*→*right_x*, *p*→*x_coord*, *r1*);
      take_fraction(*r1*, *mp*→*delta_y*[0], *ff*);
      set_number_from_addition(*p*→*right_y*, *p*→*y_coord*, *r1*);
    }
    if (*number_unity*(*lt*)) {
      mp_number *arg2*;

      new_number(*arg2*);
      if (*number_nonnegative*(*mp*→*delta_x*[0])) {
        set_number_from_addition(*arg2*, *mp*→*delta_x*[0], *epsilon_t*);
      }

```
    else {
      set_number_from_substraction(arg2, mp→delta_x[0], epsilon_t);
    }
    number_int_div(arg2, 3);
    set_number_from_substraction(q→left_x, q→x_coord, arg2);
    if (number_nonnegative(mp→delta_y[0])) {
      set_number_from_addition(arg2, mp→delta_y[0], epsilon_t);
    }
    else {
      set_number_from_substraction(arg2, mp→delta_y[0], epsilon_t);
    }
    number_int_div(arg2, 3);
    set_number_from_substraction(q→left_y, q→y_coord, arg2);
    free_number(arg2);
  }
  else {
    mp_number arg2, r1;

    new_fraction(r1);
    new_number(arg2);
    number_clone(arg2, lt);
    number_multiply_int(arg2, 3);
    make_fraction(ff, unity_t, arg2);     /* β/3 */
    free_number(arg2);
    take_fraction(r1, mp→delta_x[0], ff);
    set_number_from_substraction(q→left_x, q→x_coord, r1);
    take_fraction(r1, mp→delta_y[0], ff);
    set_number_from_substraction(q→left_y, q→y_coord, r1);
    free_number(r1);
  }
  free_number(ff);
  free_number(lt);
  free_number(rt);
  return;
}
```

This code is used in section 351.

375.   Various subroutines that are useful for the new (1.770) exported api for solving path choices

#**define** TOO_LARGE(a) $(fabs((a)) > 4096.0)$
#**define** PI   3.14159265358979323846264338327950288841971
  static int *out_of_range*(**MP** *mp*, **double** *a*)
  {
    mp_number *t*;
    new_number(*t*);
    set_number_from_double(*t*, *fabs*(*a*));
    if (*number_greaterequal*(*t*, *inf_t*)) {
      free_number(*t*);
      return 1;
    }
    free_number(*t*);
    return 0;
  }

  static int *mp_link_knotpair*(**MP** *mp*, **mp_knot** *p*, **mp_knot** *q*);

  static int *mp_link_knotpair*(**MP** *mp*, **mp_knot** *p*, **mp_knot** *q*)
  {
    if ($p \equiv \Lambda \vee q \equiv \Lambda$) **return** 0;
    p→*next* = *q*;
    set_number_from_double(*p*→*right_tension*, 1.0);
    if (*mp_right_type*(*p*) ≡ *mp_endpoint*) {
      mp_right_type(*p*) = *mp_open*;
    }
    set_number_from_double(*q*→*left_tension*, 1.0);
    if (*mp_left_type*(*q*) ≡ *mp_endpoint*) {
      mp_left_type(*q*) = *mp_open*;
    }
    return 1;
  }

  int *mp_close_path_cycle*(**MP** *mp*, **mp_knot** *p*, **mp_knot** *q*)
  {
    return *mp_link_knotpair*(*mp*, *p*, *q*);
  }

  int *mp_close_path*(**MP** *mp*, **mp_knot** *q*, **mp_knot** *first*)
  {
    if ($q \equiv \Lambda \vee first \equiv \Lambda$) **return** 0;
    q→*next* = *first*;
    mp_right_type(*q*) = *mp_endpoint*;
    set_number_from_double(*q*→*right_tension*, 1.0);
    mp_left_type(*first*) = *mp_endpoint*;
    set_number_from_double(*first*→*left_tension*, 1.0);
    return 1;
  }

  mp_knot *mp_create_knot*(**MP** *mp*)
  {
    mp_knot *q* = *mp_new_knot*(*mp*);
    mp_left_type(*q*) = *mp_endpoint*;
    mp_right_type(*q*) = *mp_endpoint*;
    return *q*;

```
  }
  int mp_set_knot(MP mp, mp_knot p, double x, double y)
  {
     if (out_of_range(mp, x)) return 0;
     if (out_of_range(mp, y)) return 0;
     if (p ≡ Λ) return 0;
     set_number_from_double(p⃗x_coord, x);
     set_number_from_double(p⃗y_coord, y);
     return 1;
  }
  mp_knot mp_append_knot(MP mp, mp_knot p, double x, double y)
  {
     mp_knot q = mp_create_knot(mp);
     if (q ≡ Λ) return Λ;
     if (¬mp_set_knot(mp, q, x, y)) {
        free(q);
        return Λ;
     }
     if (p ≡ Λ) return q;
     if (¬mp_link_knotpair(mp, p, q)) {
        free(q);
        return Λ;
     }
     return q;
  }
  int mp_set_knot_curl(MP mp, mp_knot q, double value)
  {
     if (q ≡ Λ) return 0;
     if (TOO_LARGE(value)) return 0;
     mp_right_type(q) = mp_curl;
     set_number_from_double(q⃗right_curl, value);
     if (mp_left_type(q) ≡ mp_open) {
        mp_left_type(q) = mp_curl;
        set_number_from_double(q⃗left_curl, value);
     }
     return 1;
  }
  int mp_set_knot_left_curl(MP mp, mp_knot q, double value)
  {
     if (q ≡ Λ) return 0;
     if (TOO_LARGE(value)) return 0;
     mp_left_type(q) = mp_curl;
     set_number_from_double(q⃗left_curl, value);
     if (mp_right_type(q) ≡ mp_open) {
        mp_right_type(q) = mp_curl;
        set_number_from_double(q⃗right_curl, value);
     }
     return 1;
  }
  int mp_set_knot_right_curl(MP mp, mp_knot q, double value)
  {
```

```
  if (q ≡ Λ) return 0;
  if (TOO_LARGE(value)) return 0;
  mp_right_type(q) = mp_curl;
  set_number_from_double(q⃗right_curl, value);
  if (mp_left_type(q) ≡ mp_open) {
    mp_left_type(q) = mp_curl;
    set_number_from_double(q⃗left_curl, value);
  }
  return 1;
}
int mp_set_knotpair_curls(MP mp, mp_knot p, mp_knot q, double t1, double t2)
{
  if (p ≡ Λ ∨ q ≡ Λ) return 0;
  if (mp_set_knot_curl(mp, p, t1)) return mp_set_knot_curl(mp, q, t2);
  return 0;
}
int mp_set_knotpair_tensions(MP mp, mp_knot p, mp_knot q, double t1, double t2)
{
  if (p ≡ Λ ∨ q ≡ Λ) return 0;
  if (TOO_LARGE(t1)) return 0;
  if (TOO_LARGE(t2)) return 0;
  if ((fabs(t1) < 0.75)) return 0;
  if ((fabs(t2) < 0.75)) return 0;
  set_number_from_double(p⃗right_tension, t1);
  set_number_from_double(q⃗left_tension, t2);
  return 1;
}
int mp_set_knot_left_tension(MP mp, mp_knot p, double t1)
{
  if (p ≡ Λ) return 0;
  if (TOO_LARGE(t1)) return 0;
  if ((fabs(t1) < 0.75)) return 0;
  set_number_from_double(p⃗left_tension, t1);
  return 1;
}
int mp_set_knot_right_tension(MP mp, mp_knot p, double t1)
{
  if (p ≡ Λ) return 0;
  if (TOO_LARGE(t1)) return 0;
  if ((fabs(t1) < 0.75)) return 0;
  set_number_from_double(p⃗right_tension, t1);
  return 1;
}
int mp_set_knotpair_controls(MP mp, mp_knot p, mp_knot q, double x1, double y1, double
        x2, double y2)
{
  if (p ≡ Λ ∨ q ≡ Λ) return 0;
  if (out_of_range(mp, x1)) return 0;
  if (out_of_range(mp, y1)) return 0;
  if (out_of_range(mp, x2)) return 0;
  if (out_of_range(mp, y2)) return 0;
```

```
    mp_right_type(p) = mp_explicit;
    set_number_from_double(p⃗right_x, x1);
    set_number_from_double(p⃗right_y, y1);
    mp_left_type(q) = mp_explicit;
    set_number_from_double(q⃗left_x, x2);
    set_number_from_double(q⃗left_y, y2);
    return 1;
}
int mp_set_knot_left_control(MP mp, mp_knot p, double x1, double y1)
{
    if (p ≡ Λ) return 0;
    if (out_of_range(mp, x1)) return 0;
    if (out_of_range(mp, y1)) return 0;
    mp_left_type(p) = mp_explicit;
    set_number_from_double(p⃗left_x, x1);
    set_number_from_double(p⃗left_y, y1);
    return 1;
}
int mp_set_knot_right_control(MP mp, mp_knot p, double x1, double y1)
{
    if (p ≡ Λ) return 0;
    if (out_of_range(mp, x1)) return 0;
    if (out_of_range(mp, y1)) return 0;
    mp_right_type(p) = mp_explicit;
    set_number_from_double(p⃗right_x, x1);
    set_number_from_double(p⃗right_y, y1);
    return 1;
}
int mp_set_knot_direction(MP mp, mp_knot q, double x, double y)
{
    double value = 0;
    if (q ≡ Λ) return 0;
    if (TOO_LARGE(x)) return 0;
    if (TOO_LARGE(y)) return 0;
    if (¬(x ≡ 0 ∧ y ≡ 0)) value = atan2(y, x) * (180.0/PI) * 16.0;
    mp_right_type(q) = mp_given;
    set_number_from_double(q⃗right_curl, value);
    if (mp_left_type(q) ≡ mp_open) {
        mp_left_type(q) = mp_given;
        set_number_from_double(q⃗left_curl, value);
    }
    return 1;
}
int mp_set_knotpair_directions(MP mp, mp_knot p, mp_knot q, double x1, double y1, double
        x2, double y2)
{
    if (p ≡ Λ ∨ q ≡ Λ) return 0;
    if (mp_set_knot_direction(mp, p, x1, y1)) return mp_set_knot_direction(mp, q, x2, y2);
    return 0;
}
```

376.

```
static int path_needs_fixing(mp_knot source);

static int path_needs_fixing(mp_knot source)
{
    mp_knot sourcehead = source;
    do {
        source = source⃗next;
    } while (source ∧ source ≠ sourcehead);
    if (¬source) {
        return 1;
    }
    return 0;
}

int mp_solve_path(MP mp, mp_knot first)
{
    int saved_arith_error = mp⃗arith_error;
    jmp_buf *saved_jump_buf = mp⃗jump_buf;
    int retval = 1;
    if (first ≡ Λ) return 0;
    if (path_needs_fixing(first)) return 0;
    mp⃗jump_buf = malloc(sizeof(jmp_buf));
    if (mp⃗jump_buf ≡ Λ ∨ setjmp(*(mp⃗jump_buf)) ≠ 0) {
        return 0;
    }
    mp⃗arith_error = 0;
    mp_make_choices(mp, first);
    if (mp⃗arith_error) retval = 0;
    mp⃗arith_error = saved_arith_error;
    free(mp⃗jump_buf);
    mp⃗jump_buf = saved_jump_buf;
    return retval;
}

void mp_free_path(MP mp, mp_knot p)
{
    mp_toss_knot_list(mp, p);
}
```

377.   ⟨ Exported function headers 18 ⟩ +≡

   int *mp_close_path_cycle*(**MP** *mp*, **mp_knot** *p*, **mp_knot** *q*);

   int *mp_close_path*(**MP** *mp*, **mp_knot** *q*, **mp_knot** *first*);

   mp_knot *mp_create_knot*(**MP** *mp*);

   int *mp_set_knot*(**MP** *mp*, **mp_knot** *p*, **double** *x*, **double** *y*);

   mp_knot *mp_append_knot*(**MP** *mp*, **mp_knot** *p*, **double** *x*, **double** *y*);

   int *mp_set_knot_curl*(**MP** *mp*, **mp_knot** *q*, **double** *value*);

   int *mp_set_knot_left_curl*(**MP** *mp*, **mp_knot** *q*, **double** *value*);

   int *mp_set_knot_right_curl*(**MP** *mp*, **mp_knot** *q*, **double** *value*);

   int *mp_set_knotpair_curls*(**MP** *mp*, **mp_knot** *p*, **mp_knot** *q*, **double** *t1*, **double** *t2*);

   int *mp_set_knotpair_tensions*(**MP** *mp*, **mp_knot** *p*, **mp_knot** *q*, **double** *t1*, **double** *t2*);

   int *mp_set_knot_left_tension*(**MP** *mp*, **mp_knot** *p*, **double** *t1*);

   int *mp_set_knot_right_tension*(**MP** *mp*, **mp_knot** *p*, **double** *t1*);

   int *mp_set_knot_left_control*(**MP** *mp*, **mp_knot** *p*, **double** *t1*, **double** *t2*);

   int *mp_set_knot_right_control*(**MP** *mp*, **mp_knot** *p*, **double** *t1*, **double** *t2*);

   int *mp_set_knotpair_controls*(**MP** *mp*, **mp_knot** *p*, **mp_knot** *q*, **double** *x1*, **double** *y1*, **double** *x2*, **double** *y2*);

   int *mp_set_knot_direction*(**MP** *mp*, **mp_knot** *q*, **double** *x*, **double** *y*);

   int *mp_set_knotpair_directions*(**MP** *mp*, **mp_knot** *p*, **mp_knot** *q*, **double** *x1*, **double** *y1*, **double** *x2*, **double** *y2*);

   int *mp_solve_path*(**MP** *mp*, **mp_knot** *first*);

   void *mp_free_path*(**MP** *mp*, **mp_knot** *p*);

378.   Simple accessors for **mp_knot**.

mp_number $mp\_knot\_x\_coord$(**MP** mp, **mp_knot** p)
{
    return $p\rightarrow x\_coord$;
}
mp_number $mp\_knot\_y\_coord$(**MP** mp, **mp_knot** p)
{
    return $p\rightarrow y\_coord$;
}
mp_number $mp\_knot\_left\_x$(**MP** mp, **mp_knot** p)
{
    return $p\rightarrow left\_x$;
}
mp_number $mp\_knot\_left\_y$(**MP** mp, **mp_knot** p)
{
    return $p\rightarrow left\_y$;
}
mp_number $mp\_knot\_right\_x$(**MP** mp, **mp_knot** p)
{
    return $p\rightarrow right\_x$;
}
mp_number $mp\_knot\_right\_y$(**MP** mp, **mp_knot** p)
{
    return $p\rightarrow right\_y$;
}
int $mp\_knot\_right\_type$(**MP** mp, **mp_knot** p)
{
    return $mp\_right\_type(p)$;
}
int $mp\_knot\_left\_type$(**MP** mp, **mp_knot** p)
{
    return $mp\_left\_type(p)$;
}
mp_knot mp_knot_next(**MP** mp, **mp_knot** p)
{
    return $p\rightarrow next$;
}
double $mp\_number\_as\_double$(**MP** mp, **mp_number** n)
{
    return $number\_to\_double(n)$;
}

379.   ⟨ Exported function headers 18 ⟩ +≡

#define *mp_knot_left_curl*   *mp_knot_left_x*
#define *mp_knot_left_given*   *mp_knot_left_x*
#define *mp_knot_left_tension*   *mp_knot_left_y*
#define *mp_knot_right_curl*   *mp_knot_right_x*
#define *mp_knot_right_given*   *mp_knot_right_x*
#define *mp_knot_right_tension*   *mp_knot_right_y*
  mp_number *mp_knot_x_coord* (**MP** *mp*, **mp_knot** *p*);
  mp_number *mp_knot_y_coord* (**MP** *mp*, **mp_knot** *p*);
  mp_number *mp_knot_left_x* (**MP** *mp*, **mp_knot** *p*);
  mp_number *mp_knot_left_y* (**MP** *mp*, **mp_knot** *p*);
  mp_number *mp_knot_right_x* (**MP** *mp*, **mp_knot** *p*);
  mp_number *mp_knot_right_y* (**MP** *mp*, **mp_knot** *p*);
  int *mp_knot_right_type* (**MP** *mp*, **mp_knot** *p*);
  int *mp_knot_left_type* (**MP** *mp*, **mp_knot** *p*);
  mp_knot *mp_knot_next* (**MP** *mp*, **mp_knot** *p*);
  double *mp_number_as_double* (**MP** *mp*, **mp_number** *n*);

380.   Measuring paths.   METAPOST's **llcorner**, **lrcorner**, **ulcorner**, and **urcorner** operators allow the user to measure the bounding box of anything that can go into a picture. It's easy to get rough bounds on the x and y extent of a path by just finding the bounding box of the knots and the control points. We need a more accurate version of the bounding box, but we can still use the easy estimate to save time by focusing on the interesting parts of the path.

381.   Computing an accurate bounding box involves a theme that will come up again and again. Given a Bernshteĭn polynomial

$$B(z_0, z_1, \ldots, z_n; t) = \sum_k \binom{n}{k} t^k (1 - t)^{n-k} z_k,$$

we can conveniently bisect its range as follows:

1) Let $z_k^{(0)} = z_k$, for $0 \le k \le n$.

2) Let $z_k^{(j+1)} = \frac{1}{2}(z_k^{(j)} + z_{k+1}^{(j)})$, for $0 \le k < n - j$, for $0 \le j < n$.

Then

$$B(z_0, z_1, \ldots, z_n; t) = B(z_0^{(0)}, z_0^{(1)}, \ldots, z_0^{(n)}; 2t) = B(z_0^{(n)}, z_1^{(n-1)}, \ldots, z_n^{(0)}; 2t - 1).$$

This formula gives us the coefficients of polynomials to use over the ranges $0 Ł t Ł \frac{1}{2}$ and $\frac{1}{2} Ł t Ł 1$.

382.   Here is a routine that computes the x or y coordinate of the point on a cubic corresponding to the *fraction* value t.

```
static void mp_eval_cubic(MP mp, mp_number *r, mp_knot p, mp_knot q, quarterword
        c, mp_number t)
{
  mp_number x1, x2, x3;      /* intermediate values */
  new_number(x1);
  new_number(x2);
  new_number(x3);
  if (c ≡ mp_x_code) {
    set_number_from_of_the_way(x1, t, p→x_coord, p→right_x);
    set_number_from_of_the_way(x2, t, p→right_x, q→left_x);
    set_number_from_of_the_way(x3, t, q→left_x, q→x_coord);
  }
  else {
    set_number_from_of_the_way(x1, t, p→y_coord, p→right_y);
    set_number_from_of_the_way(x2, t, p→right_y, q→left_y);
    set_number_from_of_the_way(x3, t, q→left_y, q→y_coord);
  }
  set_number_from_of_the_way(x1, t, x1, x2);
  set_number_from_of_the_way(x2, t, x2, x3);
  set_number_from_of_the_way(*r, t, x1, x2);
  free_number(x1);
  free_number(x2);
  free_number(x3);
}
```

383.   The actual bounding box information is stored in global variables. Since it is convenient to address the x and y information separately, we define arrays indexed by x_code .. y_code and use macros to give them more convenient names.

⟨Types in the outer block 33⟩ +≡
  enum mp_bb_code {
    $mp\_x\_code = 0$,   /∗ index for $minx$ and $maxx$ ∗/
    mp_y_code /∗ index for $miny$ and $maxy$ ∗/
  };

384.

#define mp_minx $mp{\rightarrow}bbmin[mp\_x\_code]$
#define mp_maxx $mp{\rightarrow}bbmax[mp\_x\_code]$
#define mp_miny $mp{\rightarrow}bbmin[mp\_y\_code]$
#define mp_maxy $mp{\rightarrow}bbmax[mp\_y\_code]$

⟨Global variables 14⟩ +≡
  mp_number $bbmin[mp\_y\_code + 1]$;
  mp_number $bbmax[mp\_y\_code + 1]$;
    /∗ the result of procedures that compute bounding box information ∗/

385.   ⟨Initialize table entries 182⟩ +≡
  {
    int i;
    for $(i = 0;\ i \leq mp\_y\_code;\ i{+}{+})$ {
      $new\_number(mp{\rightarrow}bbmin[i])$;
      $new\_number(mp{\rightarrow}bbmax[i])$;
    }
  }

386.   ⟨Dealloc variables 27⟩ +≡
  {
    int i;
    for $(i = 0;\ i \leq mp\_y\_code;\ i{+}{+})$ {
      $free\_number(mp{\rightarrow}bbmin[i])$;
      $free\_number(mp{\rightarrow}bbmax[i])$;
    }
  }

387.   Now we're ready for the key part of the bounding box computation. The *bound_cubic* procedure updates *bbmin*[c] and *bbmax*[c] based on

$$B(\mathit{knot\_coord}(p), \mathit{right\_coord}(p), \mathit{left\_coord}(q), \mathit{knot\_coord}(q); t)$$

for $0 < t \leq 1$. In other words, the procedure adjusts the bounds to accommodate *knot_coord*(q) and any extremes over the range $0 < t < 1$. The c parameter is *x_code* or *y_code*.

```
static void mp_bound_cubic(MP mp, mp_knot p, mp_knot q, quarterword c)
{
  boolean wavy;       /* whether we need to look for extremes */
  mp_number del1, del2, del3, del, dmax;
    /* proportional to the control points of a quadratic derived from a cubic */
  mp_number t, tt;      /* where a quadratic crosses zero */
  mp_number x;      /* a value that bbmin[c] and bbmax[c] must accommodate */
  new_number(x);
  new_fraction(t);
  new_fraction(tt);
  if (c ≡ mp_x_code) {
    number_clone(x, q→x_coord);
  }
  else {
    number_clone(x, q→y_coord);
  }
  new_number(del1);
  new_number(del2);
  new_number(del3);
  new_number(del);
  new_number(dmax);
  ⟨Adjust bbmin[c] and bbmax[c] to accommodate x 388⟩;
  ⟨Check the control points against the bounding box and set wavy: = true if any of them lie
      outside 389⟩;
  if (wavy) {
    if (c ≡ mp_x_code) {
      set_number_from_substraction(del1, p→right_x, p→x_coord);
      set_number_from_substraction(del2, q→left_x, p→right_x);
      set_number_from_substraction(del3, q→x_coord, q→left_x);
    }
    else {
      set_number_from_substraction(del1, p→right_y, p→y_coord);
      set_number_from_substraction(del2, q→left_y, p→right_y);
      set_number_from_substraction(del3, q→y_coord, q→left_y);
    }
    ⟨Scale up del1, del2, and del3 for greater accuracy; also set del to the first nonzero element of
        (del1, del2, del3) 390⟩;
    if (number_negative(del)) {
      number_negate(del1);
      number_negate(del2);
      number_negate(del3);
    }
    crossing_point(t, del1, del2, del3);
    if (number_less(t, fraction_one_t)) {
      ⟨Test the extremes of the cubic against the bounding box 391⟩;
```

```
    }
  }
  free_number (del3 );
  free_number (del2 );
  free_number (del1 );
  free_number (del );
  free_number (dmax );
  free_number (x);
  free_number (t);
  free_number (tt );
}
```

388. ⟨Adjust *bbmin*[*c*] and *bbmax*[*c*] to accommodate *x* 388⟩ ≡
  if (*number_less*(*x*, *mp*→*bbmin*[*c*])) *number_clone*(*mp*→*bbmin*[*c*], *x*);
  if (*number_greater*(*x*, *mp*→*bbmax*[*c*])) *number_clone*(*mp*→*bbmax*[*c*], *x*)

This code is used in sections 387, 391, and 392.

389. ⟨Check the control points against the bounding box and set *wavy*: = *true* if any of them lie
      outside 389⟩ ≡
  wavy = *true*;
  if (*c* ≡ *mp_x_code*) {
    if (*number_lessequal*(*mp*→*bbmin*[*c*], *p*→*right_x*))
      if (*number_lessequal*(*p*→*right_x*, *mp*→*bbmax*[*c*]))
        if (*number_lessequal*(*mp*→*bbmin*[*c*], *q*→*left_x*))
          if (*number_lessequal*(*q*→*left_x*, *mp*→*bbmax*[*c*])) *wavy* = *false*;
  }
  else {
    if (*number_lessequal*(*mp*→*bbmin*[*c*], *p*→*right_y*))
      if (*number_lessequal*(*p*→*right_y*, *mp*→*bbmax*[*c*]))
        if (*number_lessequal*(*mp*→*bbmin*[*c*], *q*→*left_y*))
          if (*number_lessequal*(*q*→*left_y*, *mp*→*bbmax*[*c*])) *wavy* = *false*;
  }

This code is used in section 387.

390.   If $del1 = del2 = del3 = 0$, it's impossible to obey the title of this section. We just set $del = 0$ in that case.

$\langle$ Scale up $del1$, $del2$, and $del3$ for greater accuracy; also set del to the first nonzero element of
        $(del1, del2, del3)$ 390 $\rangle \equiv$
  if $(number\_nonzero(del1))$ {
    $number\_clone(del, del1);$
  }
  else if $(number\_nonzero(del2))$ {
    $number\_clone(del, del2);$
  }
  else {
    $number\_clone(del, del3);$
  }
  if $(number\_nonzero(del))$ {
    mp_number $absval1;$

    $new\_number(absval1);$
    $number\_clone(dmax, del1);$
    $number\_abs(dmax);$
    $number\_clone(absval1, del2);$
    $number\_abs(absval1);$
    if $(number\_greater(absval1, dmax))$ {
      $number\_clone(dmax, absval1);$
    }
    $number\_clone(absval1, del3);$
    $number\_abs(absval1);$
    if $(number\_greater(absval1, dmax))$ {
      $number\_clone(dmax, absval1);$
    }
    while $(number\_less(dmax, fraction\_half\_t))$ {
      $number\_double(dmax);$
      $number\_double(del1);$
      $number\_double(del2);$
      $number\_double(del3);$
    }
    $free\_number(absval1);$
  }
This code is used in section 387.

391.   Since *crossing_point* has tried to choose *t* so that $B(del1, del2, del3; \tau)$ crosses zero at $\tau = t$ with negative slope, the value of *del2* computed below should not be positive. But rounding error could make it slightly positive in which case we must cut it to zero to avoid confusion.

⟨ Test the extremes of the cubic against the bounding box 391 ⟩ ≡
```
{
    mp_eval_cubic(mp, &x, p, q, c, t);
    ⟨ Adjust bbmin[c] and bbmax[c] to accommodate x 388 ⟩;
    set_number_from_of_the_way(del2, t, del2, del3);
        /* now 0, del2, del3 represent the derivative on the remaining interval */
    if (number_positive(del2)) set_number_to_zero(del2);
    {
        mp_number arg2, arg3;

        new_number(arg2);
        new_number(arg3);
        number_clone(arg2, del2);
        number_negate(arg2);
        number_clone(arg3, del3);
        number_negate(arg3);
        crossing_point(tt, zero_t, arg2, arg3);
        free_number(arg2);
        free_number(arg3);
    }
    if (number_less(tt, fraction_one_t)) {
        ⟨ Test the second extreme against the bounding box 392 ⟩;
    }
}
```
This code is used in section 387.

392.   ⟨ Test the second extreme against the bounding box 392 ⟩ ≡
```
{
    mp_number arg;

    new_number(arg);
    set_number_from_of_the_way(arg, t, tt, fraction_one_t);
    mp_eval_cubic(mp, &x, p, q, c, arg);
    free_number(arg);
    ⟨ Adjust bbmin[c] and bbmax[c] to accommodate x 388 ⟩;
}
```
This code is used in section 391.

393.   Finding the bounding box of a path is basically a matter of applying *bound_cubic* twice for each pair of adjacent knots.

> **static void** *mp_path_bbox*(**MP** *mp*, **mp_knot** *h*)
> {
>   **mp_knot** *p*, *q*;   /∗ a pair of adjacent knots ∗/
>   *number_clone*(*mp_minx*, *h⇾x_coord*);
>   *number_clone*(*mp_miny*, *h⇾y_coord*);
>   *number_clone*(*mp_maxx*, *mp_minx*);
>   *number_clone*(*mp_maxy*, *mp_miny*);
>   *p* = *h*;
>   **do** {
>   **if** (*mp_right_type*(*p*) ≡ *mp_endpoint*) **return**;
>   *q* = *mp_next_knot*(*p*);
>   *mp_bound_cubic*(*mp*, *p*, *q*, *mp_x_code*);
>   *mp_bound_cubic*(*mp*, *p*, *q*, *mp_y_code*);
>   *p* = *q*;
>   } **while** (*p* ≠ *h*);
> }

394.   Another important way to measure a path is to find its arc length. This is best done by using the general bisection algorithm to subdivide the path until obtaining "well behaved" subpaths whose arc lengths can be approximated by simple means.

Since the arc length is the integral with respect to time of the magnitude of the velocity, it is natural to use Simpson's rule for the approximation. If $\dot{B}(t)$ is the spline velocity, Simpson's rule gives

$$\frac{|\dot{B}(0)| + 4|\dot{B}(\frac{1}{2})| + |\dot{B}(1)|}{6}$$

for the arc length of a path of length 1. For a cubic spline $B(z_0, z_1, z_2, z_3; t)$, the time derivative $\dot{B}(t)$ is $3B(dz_0, dz_1, dz_2; t)$, where $dz_i = z_{i+1} - z_i$. Hence the arc length approximation is

$$\frac{|dz_0|}{2} + 2|dz_{02}| + \frac{|dz_2|}{2},$$

where

$$dz_{02} = \frac{1}{2}\left(\frac{dz_0 + dz_1}{2} + \frac{dz_1 + dz_2}{2}\right)$$

is the result of the bisection algorithm.

395.   The remaining problem is how to decide when a subpath is "well behaved." This could be done via the theoretical error bound for Simpson's rule, but this is impractical because it requires an estimate of the fourth derivative of the quantity being integrated. It is much easier to just perform a bisection step and see how much the arc length estimate changes. Since the error for Simpson's rule is proportional to the fourth power of the sample spacing, the remaining error is typically about $\frac{1}{16}$ of the amount of the change. We say "typically" because the error has a pseudo-random behavior that could cause the two estimates to agree when each contain large errors.

To protect against disasters such as undetected cusps, the bisection process should always continue until all the dz_i vectors belong to a single 90° sector. This ensures that no point on the spline can have velocity less than 70% of the minimum of $|dz_0|$, $|dz_1|$ and $|dz_2|$. If such a spline happens to produce an erroneous arc length estimate that is little changed by bisection, the amount of the error is likely to be fairly small. We will try to arrange things so that freak accidents of this type do not destroy the inverse relationship between the **arclength** and **arctime** operations.

396.   The **arclength** and **arctime** operations are both based on a recursive function that finds the arc length of a cubic spline given dz_0, dz_1, dz_2. This *arc_test* routine also takes an arc length goal *a_goal* and returns the time when the arc length reaches *a_goal* if there is such a time. Thus the return value is either an arc length less than *a_goal* or, if the arc length would be at least *a_goal*, it returns a time value decreased by *two*. This allows the caller to use the sign of the result to distinguish between arc lengths and time values. On certain types of overflow, it is possible for *a_goal* and the result of *arc_test* both to be `EL_GORDO`. Otherwise, the result is always less than *a_goal*.

Rather than halving the control point coordinates on each recursive call to *arc_test*, it is better to keep them proportional to velocity on the original curve and halve the results instead. This means that recursive calls can potentially use larger error tolerances in their arc length estimates. How much larger depends on to what extent the errors behave as though they are independent of each other. To save computing time, we use optimistic assumptions and increase the tolerance by a factor of about $\sqrt{2}$ for each recursive call.

In addition to the tolerance parameter, *arc_test* should also have parameters for $\frac{1}{3}|\dot{B}(0)|$, $\frac{2}{3}|\dot{B}(\frac{1}{2})|$, and $\frac{1}{3}|\dot{B}(1)|$. These quantities are relatively expensive to compute and they are needed in different instances of *arc_test*.

> **static void** *mp_arc_test*(**MP** *mp*, **mp_number** *∗ret*, **mp_number** *dx0*, **mp_number** *dy0*, **mp_number** *dx1*, **mp_number** *dy1*, **mp_number** *dx2*, **mp_number** *dy2*, **mp_number** *v0*, **mp_number** *v02*, **mp_number** *v2*, **mp_number** *a_goal*, **mp_number** *tol_orig*)
> {
>   **boolean** *simple*;　　/∗ are the control points confined to a 90° sector? ∗/
>   **mp_number** *dx01*, *dy01*, *dx12*, *dy12*, *dx02*, *dy02*;　　/∗ bisection results ∗/
>   **mp_number** *v002*, *v022*;　　/∗ twice the velocity magnitudes at $t = \frac{1}{4}$ and $t = \frac{3}{4}$ ∗/
>   **mp_number** *arc*;　　/∗ best arc length estimate before recursion ∗/
>   **mp_number** *arc1*;　　/∗ arc length estimate for the first half ∗/
>   **mp_number** *simply*;
>   **mp_number** *tol*;
>   *new_number*(*arc*);
>   *new_number*(*arc1*);
>   *new_number*(*dx01*);
>   *new_number*(*dy01*);
>   *new_number*(*dx12*);
>   *new_number*(*dy12*);
>   *new_number*(*dx02*);
>   *new_number*(*dy02*);
>   *new_number*(*v002*);
>   *new_number*(*v022*);
>   *new_number*(*simply*);
>   *new_number*(*tol*);
>   *number_clone*(*tol*, *tol_orig*);
>   ⟨ Bisect the Bézier quadratic given by *dx0*, *dy0*, *dx1*, *dy1*, *dx2*, *dy2* 400 ⟩;
>   ⟨ Initialize *v002*, *v022*, and the arc length estimate *arc*; if it overflows set *arc_test* and **return** 401 ⟩;
>   ⟨ Test if the control points are confined to one quadrant or rotating them 45° would put them in one quadrant. Then set *simple* appropriately 402 ⟩;
>   *set_number_from_addition*(*simply*, *v0*, *v2*);
>   *number_halfp*(*simply*);
>   *number_negate*(*simply*);
>   *number_add*(*simply*, *arc*);
>   *number_substract*(*simply*, *v02*);
>   *number_abs*(*simply*);
>   **if** (*simple* ∧ *number_lessequal*(*simply*, *tol*)) {
>   **if** (*number_less*(*arc*, *a_goal*)) {

$number\_clone(*ret, arc)$;
}
else {
⟨ Estimate when the arc length reaches a_goal and set arc_test to that time minus two 403 ⟩;
}
}
else {
⟨ Use one or two recursive calls to compute the arc_test function 397 ⟩;
}
DONE: $free\_number(arc)$;
  $free\_number(arc1)$;
  $free\_number(dx01)$;
  $free\_number(dy01)$;
  $free\_number(dx12)$;
  $free\_number(dy12)$;
  $free\_number(dx02)$;
  $free\_number(dy02)$;
  $free\_number(v002)$;
  $free\_number(v022)$;
  $free\_number(simply)$;
  $free\_number(tol)$;
}

397.   The *tol* value should by multiplied by $\sqrt{2}$ before making recursive calls, but 1.5 is an adequate approximation. It is best to avoid using *make_fraction* in this inner loop.

⟨ Use one or two recursive calls to compute the *arc_test* function 397 ⟩ ≡
```
{
    mp_number a_new, a_aux;      /* the sum of these gives the a_goal */
    mp_number a, b;       /* results of recursive calls */
    mp_number half_v02;       /* halfp(v02), a recursion argument */

    new_number(a_new);
    new_number(a_aux);
    new_number(half_v02);
    ⟨ Set a_new and a_aux so their sum is 2 * a_goal and a_new is as large as possible 398 ⟩;
    {
        mp_number halfp_tol;

        new_number(halfp_tol);
        number_clone(halfp_tol, tol);
        number_halfp(halfp_tol);
        number_add(tol, halfp_tol);
        free_number(halfp_tol);
    }
    number_clone(half_v02, v02);
    number_halfp(half_v02);
    new_number(a);
    mp_arc_test(mp, &a, dx0, dy0, dx01, dy01, dx02, dy02, v0, v002, half_v02, a_new, tol);
    if (number_negative(a)) {
        set_number_to_unity(*ret);
        number_double(*ret);      /* two */
        number_substract(*ret, a);      /* two - a */
        number_halfp(*ret);
        number_negate(*ret);      /* -halfp(two - a) */
    }
    else {
        ⟨ Update a_new to reduce a_new + a_aux by a 399 ⟩;
        new_number(b);
        mp_arc_test(mp, &b, dx02, dy02, dx12, dy12, dx2, dy2, half_v02, v022, v2, a_new, tol);
        if (number_negative(b)) {
            mp_number tmp;

            new_number(tmp);
            number_clone(tmp, b);
            number_negate(tmp);
            number_halfp(tmp);
            number_negate(tmp);
            number_clone(*ret, tmp);
            set_number_to_unity(tmp);
            number_halfp(tmp);
            number_substract(*ret, tmp);      /* (-(halfp(-b)) - 1/2) */
            free_number(tmp);
        }
        else {
            set_number_from_substraction(*ret, b, a);
            number_half(*ret);
            set_number_from_addition(*ret, a, *ret);      /* (a + half(b - a)) */
```

```
      }
    free_number(b);
  }
  free_number(half_v02);
  free_number(a_aux);
  free_number(a_new);
  free_number(a);
  goto DONE;
}
```

This code is used in section 396.

398. ⟨Set a_new and a_aux so their sum is $2 * a\_goal$ and a_new is as large as possible 398⟩ ≡
```
  set_number_to_inf(a_aux);
  number_substract(a_aux, a_goal);
  if (number_greater(a_goal, a_aux)) {
    set_number_from_substraction(a_aux, a_goal, a_aux);
    set_number_to_inf(a_new);
  }
  else {
    set_number_from_addition(a_new, a_goal, a_goal);
    set_number_to_zero(a_aux);
  }
```

This code is used in section 397.

399.   There is no need to maintain a_aux at this point so we use it as a temporary to force the additions and subtractions to be done in an order that avoids overflow.

⟨Update a_new to reduce $a\_new + a\_aux$ by a 399⟩ ≡
```
  if (number_greater(a, a_aux)) {
    number_substract(a_aux, a);
    number_add(a_new, a_aux);
  }
```

This code is used in section 397.

400.   This code assumes all dx and dy variables have magnitude less than fraction_four. To simplify the rest of the arc_test routine, we strengthen this assumption by requiring the norm of each (dx, dy) pair to obey this bound. Note that recursive calls will maintain this invariant.

⟨Bisect the Bézier quadratic given by dx0, dy0, dx1, dy1, dx2, dy2 400⟩ ≡
```
  set_number_from_addition(dx01, dx0, dx1);
  number_half(dx01);
  set_number_from_addition(dx12, dx1, dx2);
  number_half(dx12);
  set_number_from_addition(dx02, dx01, dx12);
  number_half(dx02);
  set_number_from_addition(dy01, dy0, dy1);
  number_half(dy01);
  set_number_from_addition(dy12, dy1, dy2);
  number_half(dy12);
  set_number_from_addition(dy02, dy01, dy12);
  number_half(dy02);
```

This code is used in section 396.

401.   We should be careful to keep $arc < \texttt{EL\_GORDO}$ so that calling arc_test with $a\_goal = \texttt{EL\_GORDO}$ is guaranteed to yield the arc length.

$\langle$ Initialize $v002$, $v022$, and the arc length estimate arc; if it overflows set arc_test and **return** $401 \rangle \equiv$
```
  {
    mp_number tmp, arg1, arg2;
    new_number(tmp);
    new_number(arg1);
    new_number(arg2);
    set_number_from_addition(arg1, dx0, dx02);
    number_half(arg1);
    number_add(arg1, dx01);
    set_number_from_addition(arg2, dy0, dy02);
    number_half(arg2);
    number_add(arg2, dy01);
    pyth_add(v002, arg1, arg2);
    set_number_from_addition(arg1, dx02, dx2);
    number_half(arg1);
    number_add(arg1, dx12);
    set_number_from_addition(arg2, dy02, dy2);
    number_half(arg2);
    number_add(arg2, dy12);
    pyth_add(v022, arg1, arg2);
    free_number(arg1);
    free_number(arg2);
    number_clone(tmp, v02);
    number_add_scaled(tmp, 2);
    number_halfp(tmp);
    set_number_from_addition(arc1, v0, tmp);
    number_halfp(arc1);
    number_substract(arc1, v002);
    number_half(arc1);
    set_number_from_addition(arc1, v002, arc1);
    set_number_from_addition(arc, v2, tmp);
    number_halfp(arc);
    number_substract(arc, v022);
    number_half(arc);
    set_number_from_addition(arc, v022, arc);     /* reuse tmp for the next if test: */
    set_number_to_inf(tmp);
    number_substract(tmp, arc1);
    if (number_less(arc, tmp)) {
      free_number(tmp);
      number_add(arc, arc1);
    }
    else {
      free_number(tmp);
      mp→arith_error = true;
      if (number_infinite(a_goal)) {
        set_number_to_inf(*ret);
      }
      else {
        set_number_to_unity(*ret);
        number_double(*ret);
```

```
        number_negate(*ret);        /* -two */
      }
      goto DONE;
    }
  }
```

This code is used in section 396.

402. ⟨Test if the control points are confined to one quadrant or rotating them 45° would put them in one quadrant. Then set *simple* appropriately 402⟩ ≡

```
  simple = ((number_nonnegative(dx0) ∧ number_nonnegative(dx1) ∧ number_nonnegative(dx2)) ∨
      (number_nonpositive(dx0) ∧ number_nonpositive(dx1) ∧ number_nonpositive(dx2)));
  if (simple) {
    simple = (number_nonnegative(dy0) ∧ number_nonnegative(dy1) ∧ number_nonnegative(dy2)) ∨
        (number_nonpositive(dy0) ∧ number_nonpositive(dy1) ∧ number_nonpositive(dy2));
  }
  if (¬simple) {
    simple = (number_greaterequal(dx0, dy0) ∧ number_greaterequal(dx1, dy1) ∧ number_greaterequal(dx2,
        dy2)) ∨ (number_lessequal(dx0, dy0) ∧ number_lessequal(dx1, dy1) ∧ number_lessequal(dx2, dy2));
    if (simple) {
      mp_number neg_dx0, neg_dx1, neg_dx2;
      new_number(neg_dx0);
      new_number(neg_dx1);
      new_number(neg_dx2);
      number_clone(neg_dx0, dx0);
      number_clone(neg_dx1, dx1);
      number_clone(neg_dx2, dx2);
      number_negate(neg_dx0);
      number_negate(neg_dx1);
      number_negate(neg_dx2);
      simple = (number_greaterequal(neg_dx0, dy0) ∧ number_greaterequal(neg_dx1,
          dy1) ∧ number_greaterequal(neg_dx2, dy2)) ∨ (number_lessequal(neg_dx0,
          dy0) ∧ number_lessequal(neg_dx1, dy1) ∧ number_lessequal(neg_dx2, dy2));
      free_number(neg_dx0);
      free_number(neg_dx1);
      free_number(neg_dx2);
    }
  }
```

This code is used in section 396.

403.   Since Simpson's rule is based on approximating the integrand by a parabola, it is appropriate to use the same approximation to decide when the integral reaches the intermediate value a_goal. At this point

$$\frac{|\dot{B}(0)|}{3} = v0\,, \qquad \frac{|\dot{B}(\frac{1}{4})|}{3} = \frac{v002}{2}\,, \qquad \frac{|\dot{B}(\frac{1}{2})|}{3} = \frac{v02}{2}\,,$$

$$\frac{|\dot{B}(\frac{3}{4})|}{3} = \frac{v022}{2}\,, \qquad \frac{|\dot{B}(1)|}{3} = v2$$

and

$$\frac{|\dot{B}(t)|}{3} \approx \begin{cases} B\left(v0\,, v002 - \frac{1}{2}v0 - \frac{1}{4}v02\,, \frac{1}{2}v02\,; 2t\right) & \text{if } t \leq \frac{1}{2} \\ B\left(\frac{1}{2}v02\,, v022 - \frac{1}{4}v02 - \frac{1}{2}v2\,, v2\,; 2t - 1\right) & \text{if } t \geq \frac{1}{2}. \end{cases} \qquad (*)$$

We can integrate $|\dot{B}(t)|$ by using

$$\int 3B(a, b, c; \tau)\, dt = \frac{B(0, a, a + b, a + b + c; \tau) + \text{constant}}{\frac{d\tau}{dt}}.$$

This construction allows us to find the time when the arc length reaches a_goal by solving a cubic equation of the form

$$B(0, a, a + b, a + b + c; \tau) = x,$$

where τ is $2t$ or $2t + 1$, x is a_goal or $a\_goal - arc1$, and a, b, and c are the Bernshteĭn coefficients from $(*)$ divided by $\frac{d\tau}{dt}$. We shall define a function $solve\_rising\_cubic$ that finds τ given a, b, c, and x.

⟨ Estimate when the arc length reaches a_goal and set arc_test to that time minus two 403 ⟩ ≡

```
{
  mp_number tmp;
  mp_number tmp2;
  mp_number tmp3;
  mp_number tmp4;
  mp_number tmp5;

  new_number(tmp);
  new_number(tmp2);
  new_number(tmp3);
  new_number(tmp4);
  new_number(tmp5);
  number_clone(tmp, v02);
  number_add_scaled(tmp, 2);
  number_half(tmp);
  number_half(tmp);      /* (v02+2) / 4 */
  if (number_lessequal(a_goal, arc1)) {
    number_clone(tmp2, v0);
    number_halfp(tmp2);
    set_number_from_substraction(tmp3, arc1, tmp2);
    number_substract(tmp3, tmp);
    mp_solve_rising_cubic(mp, &tmp5, tmp2, tmp3, tmp, a_goal);
    number_halfp(tmp5);
    set_number_to_unity(tmp3);
    number_substract(tmp5, tmp3);
    number_substract(tmp5, tmp3);
    number_clone(*ret, tmp5);
  }
  else {
    number_clone(tmp2, v2);
```

```
        number_halfp(tmp2);
        set_number_from_substraction(tmp3, arc, arc1);
        number_substract(tmp3, tmp);
        number_substract(tmp3, tmp2);
        set_number_from_substraction(tmp4, a_goal, arc1);
        mp_solve_rising_cubic(mp, &tmp5, tmp, tmp3, tmp2, tmp4);
        number_halfp(tmp5);
        set_number_to_unity(tmp2);
        set_number_to_unity(tmp3);
        number_half(tmp2);
        number_substract(tmp2, tmp3);
        number_substract(tmp2, tmp3);
        set_number_from_addition(*ret, tmp2, tmp5);
    }
    free_number(tmp);
    free_number(tmp2);
    free_number(tmp3);
    free_number(tmp4);
    free_number(tmp5);
    goto DONE;
}
```

This code is used in section 396.

404.   Here is the *solve_rising_cubic* routine that finds the time t when

$$B(0, a, a + b, a + b + c; t) = x.$$

This routine is based on *crossing_point* but is simplified by the assumptions that $B(a, b, c; t) \geq 0$ for $0 \leq t \leq 1$ and that $0 \leq x \leq a + b + c$. If rounding error causes this condition to be violated slightly, we just ignore it and proceed with binary search. This finds a time when the function value reaches x and the slope is positive.

⟨ Declarations 8 ⟩ +≡
    static void *mp_solve_rising_cubic*(**MP** *mp*, **mp_number** **ret*, **mp_number** *a*, **mp_number**
        b, **mp_number** *c*, **mp_number** *x*);

405.   **void** $mp\_solve\_rising\_cubic(\mathbf{MP}\ mp, \mathbf{mp\_number}\ *ret, \mathbf{mp\_number}\ a\_orig, \mathbf{mp\_number}$
　　　　$b\_orig, \mathbf{mp\_number}\ c\_orig, \mathbf{mp\_number}\ x\_orig)$

{

  mp_number abc;

  mp_number $a,\ b,\ c,\ x$;　　　/∗ local versions of arguments ∗/

  mp_number $ab,\ bc,\ ac$;　　　/∗ bisection results ∗/

  mp_number t;　　　/∗ $2^k + q$ where unscaled answer is in $[q2^{-k}, (q+1)2^{-k})$ ∗/

  mp_number xx;　　　/∗ temporary for updating x ∗/

  mp_number neg_x;　　　/∗ temporary for an **if** ∗/

  if $(number\_negative(a\_orig) \lor number\_negative(c\_orig))\ mp\_confusion(mp, \texttt{"rising?"})$;

  ;

  $new\_number(t)$;

  $new\_number(abc)$;

  $new\_number(a)$;

  $new\_number(b)$;

  $new\_number(c)$;

  $new\_number(x)$;

  $number\_clone(a, a\_orig)$;

  $number\_clone(b, b\_orig)$;

  $number\_clone(c, c\_orig)$;

  $number\_clone(x, x\_orig)$;

  $new\_number(ab)$;

  $new\_number(bc)$;

  $new\_number(ac)$;

  $new\_number(xx)$;

  $new\_number(neg\_x)$;

  $set\_number\_from\_addition(abc, a, b)$;

  $number\_add(abc, c)$;

  if $(number\_nonpositive(x))$ {

    $set\_number\_to\_zero(*ret)$;

  }

  else if $(number\_greaterequal(x, abc))$ {

    $set\_number\_to\_unity(*ret)$;

  }

  else {

    $number\_clone(t, epsilon\_t)$;

    ⟨ Rescale if necessary to make sure a, b, and c are all less than $\texttt{EL\_GORDO}\ div\ 3$ 407 ⟩;

    do {

      $number\_add(t, t)$;

      ⟨ Subdivide the Bézier quadratic defined by a, b, c 406 ⟩;

      $number\_clone(xx, x)$;

      $number\_substract(xx, a)$;

      $number\_substract(xx, ab)$;

      $number\_substract(xx, ac)$;

      $number\_clone(neg\_x, x)$;

      $number\_negate(neg\_x)$;

      if $(number\_less(xx, neg\_x))$ {

        $number\_double(x)$;

        $number\_clone(b, ab)$;

        $number\_clone(c, ac)$;

      }

      else {

```
        number_add(x, xx);
        number_clone(a, ac);
        number_clone(b, bc);
        number_add(t, epsilon_t);
      }
    } while (number_less(t, unity_t));
    set_number_from_substraction(*ret, t, unity_t);
  }
  free_number(abc);
  free_number(t);
  free_number(a);
  free_number(b);
  free_number(c);
  free_number(ab);
  free_number(bc);
  free_number(ac);
  free_number(xx);
  free_number(x);
  free_number(neg_x);
}
```

406.   ⟨ Subdivide the Bézier quadratic defined by a, b, c 406 ⟩ ≡
```
  set_number_from_addition(ab, a, b);
  number_half(ab);
  set_number_from_addition(bc, b, c);
  number_half(bc);
  set_number_from_addition(ac, ab, bc);
  number_half(ac);
```
This code is used in section 405.

407.   The upper bound on a, b, and c:

#define   *one_third_inf_t*   ((**math_data** ∗) *mp→math*)→*one_third_inf_t*

⟨ Rescale if necessary to make sure a, b, and c are all less than `EL_GORDO` $div\,3$ 407 ⟩ ≡
```
  while (number_greater(a, one_third_inf_t) ∨ number_greater(b, one_third_inf_t) ∨ number_greater(c,
        one_third_inf_t)) {
    number_halfp(a);
    number_half(b);
    number_halfp(c);
    number_halfp(x);
  }
```
This code is used in section 405.

408.   It is convenient to have a simpler interface to *arc_test* that requires no unnecessary arguments and
ensures that each (dx, dy) pair has length less than *fraction_four*.

```
static void mp_do_arc_test(MP mp, mp_number *ret, mp_number dx0, mp_number
        dy0, mp_number dx1, mp_number dy1, mp_number dx2, mp_number dy2, mp_number
        a_goal)
{
  mp_number v0, v1, v2;      /* length of each (dx, dy) pair */
  mp_number v02;      /* twice the norm of the quadratic at t = ½ */
  new_number(v0);
  new_number(v1);
  new_number(v2);
  pyth_add(v0, dx0, dy0);
  pyth_add(v1, dx1, dy1);
  pyth_add(v2, dx2, dy2);
  if ((number_greaterequal(v0, fraction_four_t)) ∨ (number_greaterequal(v1,
          fraction_four_t)) ∨ (number_greaterequal(v2, fraction_four_t))) {
    mp→arith_error = true;
    if (number_infinite(a_goal)) {
      set_number_to_inf(*ret);
    }
    else {
      set_number_to_unity(*ret);
      number_double(*ret);
      number_negate(*ret);
    }
  }
  else {
    mp_number arg1, arg2;

    new_number(v02);
    new_number(arg1);
    new_number(arg2);
    set_number_from_addition(arg1, dx0, dx2);
    number_half(arg1);
    number_add(arg1, dx1);
    set_number_from_addition(arg2, dy0, dy2);
    number_half(arg2);
    number_add(arg2, dy1);
    pyth_add(v02, arg1, arg2);
    free_number(arg1);
    free_number(arg2);
    mp_arc_test(mp, ret, dx0, dy0, dx1, dy1, dx2, dy2, v0, v02, v2, a_goal, arc_tol_k);
    free_number(v02);
  }
  free_number(v0);
  free_number(v1);
  free_number(v2);
}
```

409.   Now it is easy to find the arc length of an entire path.

```
static void mp_get_arc_length(MP mp, mp_number *ret, mp_knot h)
{
  mp_knot p, q;      /* for traversing the path */
  mp_number a;       /* current arc length */
  mp_number a_tot;      /* total arc length */
  mp_number arg1, arg2, arg3, arg4, arg5, arg6;
  mp_number arcgoal;

  p = h;
  new_number(a_tot);
  new_number(arg1);
  new_number(arg2);
  new_number(arg3);
  new_number(arg4);
  new_number(arg5);
  new_number(arg6);
  new_number(a);
  new_number(arcgoal);
  set_number_to_inf(arcgoal);
  while (mp_right_type(p) ≠ mp_endpoint) {
    q = mp_next_knot(p);
    set_number_from_substraction(arg1, p→right_x, p→x_coord);
    set_number_from_substraction(arg2, p→right_y, p→y_coord);
    set_number_from_substraction(arg3, q→left_x, p→right_x);
    set_number_from_substraction(arg4, q→left_y, p→right_y);
    set_number_from_substraction(arg5, q→x_coord, q→left_x);
    set_number_from_substraction(arg6, q→y_coord, q→left_y);
    mp_do_arc_test(mp, &a, arg1, arg2, arg3, arg4, arg5, arg6, arcgoal);
    slow_add(a_tot, a, a_tot);
    if (q ≡ h) break;
    else p = q;
  }
  free_number(arcgoal);
  free_number(a);
  free_number(arg1);
  free_number(arg2);
  free_number(arg3);
  free_number(arg4);
  free_number(arg5);
  free_number(arg6);
  check_arith();
  number_clone(*ret, a_tot);
  free_number(a_tot);
}
```

410.   The inverse operation of finding the time on a path h when the arc length reaches some value *arc0*
can also be accomplished via *do_arc_test*. Some care is required to handle very large times or negative times
on cyclic paths. For non-cyclic paths, *arc0* values that are negative or too large cause *get_arc_time* to return
0 or the length of path h.

If *arc0* is greater than the arc length of a cyclic path h, the result is a time value greater than the length
of the path. Since it could be much greater, we must be prepared to compute the arc length of path h and
divide this into *arc0* to find how many multiples of the length of path h to add.

$\quad$ **static void** $mp\_get\_arc\_time$(**MP** mp, **mp_number** $*ret$, **mp_knot** h, **mp_number** $arc0\_orig$)
$\quad$ {
$\qquad$ **mp_knot** p, q;   /* for traversing the path */
$\qquad$ **mp_number** t_tot;   /* accumulator for the result */
$\qquad$ **mp_number** t;   /* the result of *do_arc_test* */
$\qquad$ **mp_number** arc, $arc0$;   /* portion of *arc0* not used up so far */
$\qquad$ **mp_number** $arg1$, $arg2$, $arg3$, $arg4$, $arg5$, $arg6$;   /* *do_arc_test* arguments */
$\qquad$ **if** ($number\_negative(arc0\_orig)$) {
$\qquad\quad$ ⟨Deal with a negative *arc0_orig* value and **return** 412⟩;
$\qquad$ }
$\qquad$ $new\_number(t\_tot)$;
$\qquad$ $new\_number(arc0)$;
$\qquad$ $number\_clone(arc0, arc0\_orig)$;
$\qquad$ **if** ($number\_infinite(arc0)$) {
$\qquad\quad$ $number\_add\_scaled(arc0, -1)$;
$\qquad$ }
$\qquad$ $new\_number(arc)$;
$\qquad$ $number\_clone(arc, arc0)$;
$\qquad$ $p = h$;
$\qquad$ $new\_number(arg1)$;
$\qquad$ $new\_number(arg2)$;
$\qquad$ $new\_number(arg3)$;
$\qquad$ $new\_number(arg4)$;
$\qquad$ $new\_number(arg5)$;
$\qquad$ $new\_number(arg6)$;
$\qquad$ $new\_number(t)$;
$\qquad$ **while** (($mp\_right\_type(p) \neq mp\_endpoint$) $\land$ $number\_positive(arc)$) {
$\qquad\quad$ $q = mp\_next\_knot(p)$;
$\qquad\quad$ $set\_number\_from\_substraction(arg1, p{\rightarrow}right\_x, p{\rightarrow}x\_coord)$;
$\qquad\quad$ $set\_number\_from\_substraction(arg2, p{\rightarrow}right\_y, p{\rightarrow}y\_coord)$;
$\qquad\quad$ $set\_number\_from\_substraction(arg3, q{\rightarrow}left\_x, p{\rightarrow}right\_x)$;
$\qquad\quad$ $set\_number\_from\_substraction(arg4, q{\rightarrow}left\_y, p{\rightarrow}right\_y)$;
$\qquad\quad$ $set\_number\_from\_substraction(arg5, q{\rightarrow}x\_coord, q{\rightarrow}left\_x)$;
$\qquad\quad$ $set\_number\_from\_substraction(arg6, q{\rightarrow}y\_coord, q{\rightarrow}left\_y)$;
$\qquad\quad$ $mp\_do\_arc\_test(mp, \&t, arg1, arg2, arg3, arg4, arg5, arg6, arc)$;
$\qquad\quad$ ⟨Update *arc* and *t_tot* after *do_arc_test* has just returned t 411⟩;
$\qquad\quad$ **if** ($q \equiv h$) {
$\qquad\qquad$ ⟨Update *t_tot* and *arc* to avoid going around the cyclic path too many times but set *arith_error*:
$\qquad\qquad\quad$ = *true* and **goto** *done* on overflow 413⟩;
$\qquad\quad$ }
$\qquad\quad$ $p = q$;
$\qquad$ }
$\qquad$ $check\_arith(\,)$;
$\qquad$ $number\_clone(*ret, t\_tot)$;
$\quad$ RETURN: $free\_number(t\_tot)$;

$free\_number(t)$;
$free\_number(arc)$;
$free\_number(arc0)$;
$free\_number(arg1)$;
$free\_number(arg2)$;
$free\_number(arg3)$;
$free\_number(arg4)$;
$free\_number(arg5)$;
$free\_number(arg6)$;
}

411.   ⟨Update arc and t_tot after do_arc_test has just returned t 411⟩ ≡
  if $(number\_negative(t))$ {
    $number\_add(t\_tot, t)$;
    $number\_add(t\_tot, two\_t)$;
    $set\_number\_to\_zero(arc)$;
  }
  else {
    $number\_add(t\_tot, unity\_t)$;
    $number\_substract(arc, t)$;
  }
This code is used in section 410.

412.   ⟨Deal with a negative $arc0\_orig$ value and **return** 412⟩ ≡
  {
    if $(mp\_left\_type(h) \equiv mp\_endpoint)$ {
      $set\_number\_to\_zero(*ret)$;
    }
    else {
      mp_number neg_arc0;

      $p = mp\_htap\_ypoc(mp, h)$;
      $new\_number(neg\_arc0)$;
      $number\_clone(neg\_arc0, arc0\_orig)$;
      $number\_negate(neg\_arc0)$;
      $mp\_get\_arc\_time(mp, ret, p, neg\_arc0)$;
      $number\_negate(*ret)$;
      $mp\_toss\_knot\_list(mp, p)$;
      $free\_number(neg\_arc0)$;
    }
    $check\_arith()$;
    return;
  }
This code is used in section 410.

413.   ⟨Update *t_tot* and *arc* to avoid going around the cyclic path too many times but set *arith_error* : =
       true and **goto** *done* on overflow 413 ⟩ ≡
```
  if (number_positive(arc)) {
    mp_number n, n1, d1, v1;

    new_number(n);
    new_number(n1);
    new_number(d1);
    new_number(v1);
    set_number_from_substraction(d1, arc0, arc);      /* d1 = arc0 - arc */
    set_number_from_div(n1, arc, d1);      /* n1 = (arc / d1) */
    number_clone(n, n1);
    set_number_from_mul(n1, n1, d1);      /* n1 = (n1 * d1) */
    number_substract(arc, n1);      /* arc = arc - n1 */
    number_clone(d1, inf_t);      /* reuse d1 */
    number_clone(v1, n);      /* v1 = n */
    number_add(v1, epsilon_t);      /* v1 = n1+1 */
    set_number_from_div(d1, d1, v1);      /* d1 = EL_GORDO/v1 */
    if (number_greater(t_tot, d1)) {
      mp→arith_error = true;
      check_arith( );
      set_number_to_inf(∗ret);
      free_number(n);
      free_number(n1);
      free_number(d1);
      free_number(v1);
      goto RETURN;
    }
    set_number_from_mul(t_tot, t_tot, v1);
    free_number(n);
    free_number(n1);
    free_number(d1);
    free_number(v1);
  }
```
This code is used in section 410.

414.   Data structures for pens.   A Pen in METAPOST can be either elliptical or polygonal. Elliptical pens result in PostScript **stroke** commands, while anything drawn with a polygonal pen is converted into an area fill as described in the next part of this program. The mathematics behind this process is based on simple aspects of the theory of tracings developed by Leo Guibas, Lyle Ramshaw, and Jorge Stolfi ["A kinematic framework for computational geometry," Proc. IEEE Symp. Foundations of Computer Science **24** (1983), 100–111].

Polygonal pens are created from paths via METAPOST's **makepen** primitive. This path representation is almost sufficient for our purposes except that a pen path should always be a convex polygon with the vertices in counter-clockwise order. Since we will need to scan pen polygons both forward and backward, a pen should be represented as a doubly linked ring of knot nodes. There is room for the extra back pointer because we do not need the *mp_left_type* or *mp_right_type* fields. In fact, we don't need the *left_x*, *left_y*, *right_x*, or *right_y* fields either but we leave these alone so that certain procedures can operate on both pens and paths. In particular, pens can be copied using *copy_path* and recycled using *toss_knot_list*.

415.   The *make_pen* procedure turns a path into a pen by initializing the *prev_knot* pointers and making sure the knots form a convex polygon. Thus each cubic in the given path becomes a straight line and the control points are ignored. If the path is not cyclic, the ends are connected by a straight line.

#define   *copy_pen*(A)   *mp_make_pen*(*mp*, *mp_copy_path*(*mp*, (A)), *false*)

  static mp_knot *mp_make_pen*(**MP** *mp*, **mp_knot** *h*, **boolean** *need_hull*)
  {
    mp_knot *p*, *q*;   /∗ two consecutive knots ∗/
    q = *h*;
    do {
      p = *q*;
      q = *mp_next_knot*(*q*);
      mp_prev_knot(*q*) = *p*;
    } **while** (*q* ≠ *h*);
    if (*need_hull*) {
      h = *mp_convex_hull*(*mp*, *h*);
      ⟨ Make sure *h* isn't confused with an elliptical pen 417 ⟩;
    }
    return *h*;
  }

416.   The only information required about an elliptical pen is the overall transformation that has been applied to the original **pencircle**. Since it suffices to keep track of how the three points $(0,0)$, $(1,0)$, and $(0,1)$ are transformed, an elliptical pen can be stored in a single knot node and transformed as if it were a path.

#define $pen\_is\_elliptical(A)$ $((A) \equiv mp\_next\_knot((A)))$

```
  static mp_knot mp_get_pen_circle(MP mp, mp_number diam)
  {
    mp_knot h;      /* the knot node to return */
    h = mp_new_knot(mp);
    mp_next_knot(h) = h;
    mp_prev_knot(h) = h;
    mp_originator(h) = mp_program_code;
    set_number_to_zero(h⃗x_coord);
    set_number_to_zero(h⃗y_coord);
    number_clone(h⃗left_x, diam);
    set_number_to_zero(h⃗left_y);
    set_number_to_zero(h⃗right_x);
    number_clone(h⃗right_y, diam);
    return h;
  }
```

417.   If the polygon being returned by *make_pen* has only one vertex, it will be interpreted as an elliptical pen. This is no problem since a degenerate polygon can equally well be thought of as a degenerate ellipse. We need only initialize the *left_x*, *left_y*, *right_x*, and *right_y* fields.

⟨ Make sure h isn't confused with an elliptical pen 417 ⟩ ≡

```
  if (pen_is_elliptical(h)) {
    number_clone(h⃗left_x, h⃗x_coord);
    number_clone(h⃗left_y, h⃗y_coord);
    number_clone(h⃗right_x, h⃗x_coord);
    number_clone(h⃗right_y, h⃗y_coord);
  }
```

This code is used in section 415.

418.   Printing a polygonal pen is very much like printing a path

⟨ Declarations 8 ⟩ +≡

```
  static void mp_pr_pen(MP mp, mp_knot h);
```

419.   **void** $mp\_pr\_pen(\mathbf{MP}\ mp, \mathbf{mp\_knot}\ h)$
{
　　$\mathbf{mp\_knot}\ p,\ q;$ /∗ for list traversal ∗/
　　if $(pen\_is\_elliptical(h))$ {
　　　$\langle$ Print the elliptical pen h 421 $\rangle$;
　　}
　　else {
　　　$p = h;$
　　　do {
　　　　$mp\_print\_two(mp, p\text{→}x\_coord, p\text{→}y\_coord);$
　　　　$mp\_print\_nl(mp, "\text{␣}..\text{␣}");$
　　　　$\langle$ Advance p making sure the links are OK and **return** if there is a problem 420 $\rangle$;
　　　} **while** $(p \neq h);$
　　　$mp\_print(mp, "\text{cycle}");$
　　}
}

420. $\langle$ Advance p making sure the links are OK and **return** if there is a problem 420 $\rangle \equiv$
　$q = mp\_next\_knot(p);$
　if $((q \equiv \Lambda) \vee (mp\_prev\_knot(q) \neq p))$ {
　　$mp\_print\_nl(mp, "???");$
　　return;   /∗ this won't happen ∗/
　}
　$p = q$
This code is used in section 419.

421. $\langle$ Print the elliptical pen h 421 $\rangle \equiv$
　{
　　$\mathbf{mp\_number}\ v1;$

　　$new\_number(v1);$
　　$mp\_print(mp, "\text{pencircle␣transformed␣}(");$
　　$print\_number(h\text{→}x\_coord);$
　　$mp\_print\_char(mp, xord(\text{'},\text{'}));$
　　$print\_number(h\text{→}y\_coord);$
　　$mp\_print\_char(mp, xord(\text{'},\text{'}));$
　　$set\_number\_from\_substraction(v1, h\text{→}left\_x, h\text{→}x\_coord);$
　　$print\_number(v1);$
　　$mp\_print\_char(mp, xord(\text{'},\text{'}));$
　　$set\_number\_from\_substraction(v1, h\text{→}right\_x, h\text{→}x\_coord);$
　　$print\_number(v1);$
　　$mp\_print\_char(mp, xord(\text{'},\text{'}));$
　　$set\_number\_from\_substraction(v1, h\text{→}left\_y, h\text{→}y\_coord);$
　　$print\_number(v1);$
　　$mp\_print\_char(mp, xord(\text{'},\text{'}));$
　　$set\_number\_from\_substraction(v1, h\text{→}right\_y, h\text{→}y\_coord);$
　　$print\_number(v1);$
　　$mp\_print\_char(mp, xord(\text{'})\text{'}));$
　　$free\_number(v1);$
　}
This code is used in section 419.

422.   Here us another version of *pr_pen* that prints the pen as a diagnostic message.

⟨ Declarations 8 ⟩ +≡
  static void *mp_print_pen*(**MP** *mp*, **mp_knot** *h*, **const char** *∗s*, **boolean** *nuline*);

423.   **void** *mp_print_pen*(**MP** *mp*, **mp_knot** *h*, **const char** *∗s*, **boolean** *nuline*)
  {
     mp_print_diagnostic(*mp*, "Pen", *s*, *nuline*);
     mp_print_ln(*mp*);
     ;
     mp_pr_pen(*mp*, *h*);
     mp_end_diagnostic(*mp*, *true*);
  }

424.   Making a polygonal pen into a path involves restoring the *mp_left_type* and *mp_right_type* fields and
setting the control points so as to make a polygonal path.

  static void *mp_make_path*(**MP** *mp*, **mp_knot** *h*)
  {
     mp_knot *p*;   /∗ for traversing the knot list ∗/
     quarterword *k*;   /∗ a loop counter ∗/

     ⟨ Other local variables in *make_path* 428 ⟩;
     FUNCTION_TRACE1("make_path()\n");
     if (*pen_is_elliptical*(*h*)) {
        FUNCTION_TRACE1("make_path(elliptical)\n");
        ⟨ Make the elliptical pen *h* into a path 426 ⟩;
     }
     else {
        p = *h*;
        do {
           mp_left_type(*p*) = *mp_explicit*;
           mp_right_type(*p*) = *mp_explicit*;
           ⟨ copy the coordinates of knot *p* into its control points 425 ⟩;
           p = *mp_next_knot*(*p*);
        } **while** (*p* ≠ *h*);
     }
  }

425.   ⟨ copy the coordinates of knot *p* into its control points 425 ⟩ ≡
  number_clone(*p⇁left_x*, *p⇁x_coord*);
  number_clone(*p⇁left_y*, *p⇁y_coord*);
  number_clone(*p⇁right_x*, *p⇁x_coord*); *number_clone*(*p⇁right_y*, *p⇁y_coord*)

This code is used in section 424.

426.   We need an eight knot path to get a good approximation to an ellipse.

⟨ Make the elliptical pen h into a path 426 ⟩ ≡
```
{
    mp_number center_x , center_y ;      /* translation parameters for an elliptical pen */
    mp_number width_x , width_y ;      /* the effect of a unit change in x */
    mp_number height_x , height_y ;      /* the effect of a unit change in y */
    mp_number dx , dy ;      /* the vector from knot p to its right control point */

    new_number (center_x );
    new_number (center_y );
    new_number (width_x );
    new_number (width_y );
    new_number (height_x );
    new_number (height_y );
    new_number (dx );
    new_number (dy );
    ⟨ Extract the transformation parameters from the elliptical pen h 427 ⟩;
    p = h;
    for (k = 0; k ≤ 7; k++) {
        ⟨ Initialize p as the kth knot of a circle of unit diameter, transforming it appropriately 429 ⟩;
        if (k ≡ 7) mp_next_knot (p) = h;
        else mp_next_knot (p) = mp_new_knot (mp);
        p = mp_next_knot (p);
    }
    free_number (dx );
    free_number (dy );
    free_number (center_x );
    free_number (center_y );
    free_number (width_x );
    free_number (width_y );
    free_number (height_x );
    free_number (height_y );
}
```
This code is used in section 424.

427.   ⟨ Extract the transformation parameters from the elliptical pen h 427 ⟩ ≡
```
number_clone (center_x , h→x_coord );
number_clone (center_y , h→y_coord );
set_number_from_substraction (width_x , h→left_x , center_x );
set_number_from_substraction (width_y , h→left_y , center_y );
set_number_from_substraction (height_x , h→right_x , center_x );
set_number_from_substraction (height_y , h→right_y , center_y );
```
This code is used in section 426.

428.   ⟨ Other local variables in $make\_path$ 428 ⟩ ≡
```
    integer kk ;      /* k advanced 270° around the ring (cf. sin θ = cos(θ + 270)) */
```
This code is used in section 424.

429.   The only tricky thing here are the tables *half_cos* and *d_cos* used to find the point $k/8$ of the way around the circle and the direction vector to use there.

⟨Initialize p as the kth knot of a circle of unit diameter, transforming it appropriately 429⟩ ≡
```
kk = (k + 6) % 8;
{
   mp_number r1, r2;
   new_fraction(r1);
   new_fraction(r2);
   take_fraction(r1, mp→half_cos[k], width_x);
   take_fraction(r2, mp→half_cos[kk], height_x);
   number_add(r1, r2);
   set_number_from_addition(p→x_coord, center_x, r1);
   take_fraction(r1, mp→half_cos[k], width_y);
   take_fraction(r2, mp→half_cos[kk], height_y);
   number_add(r1, r2);
   set_number_from_addition(p→y_coord, center_y, r1);
   take_fraction(r1, mp→d_cos[kk], width_x);
   take_fraction(r2, mp→d_cos[k], height_x);
   number_clone(dx, r1);
   number_negate(dx);
   number_add(dx, r2);
   take_fraction(r1, mp→d_cos[kk], width_y);
   take_fraction(r2, mp→d_cos[k], height_y);
   number_clone(dy, r1);
   number_negate(dy);
   number_add(dy, r2);
   set_number_from_addition(p→right_x, p→x_coord, dx);
   set_number_from_addition(p→right_y, p→y_coord, dy);
   set_number_from_substraction(p→left_x, p→x_coord, dx);
   set_number_from_substraction(p→left_y, p→y_coord, dy);
   free_number(r1);
   free_number(r2);
}
mp_left_type(p) = mp_explicit;
mp_right_type(p) = mp_explicit; mp_originator(p) = mp_program_code
```
This code is used in section 426.

430.   ⟨Global variables 14⟩ +≡
mp_number *half_cos*[8];   /* $\frac{1}{2}\cos(45k)$ */
mp_number *d_cos*[8];   /* a magic constant times $\cos(45k)$ */

431.   The magic constant for d_cos is the distance between $(\frac{1}{2}, 0)$ and $(\frac{1}{4}\sqrt{2}, \frac{1}{4}\sqrt{2})$ times the result of the *velocity* function for $\theta = \phi = 22.5°$. This comes out to be

$$d = \frac{\sqrt{2 - \sqrt{2}}}{3 + 3\cos 22.5°} \approx 0.132608244919772.$$

⟨ Set initial values of key variables 38 ⟩ +≡
```
  for (k = 0; k ≤ 7; k++) {
    new_fraction(mp→half_cos[k]);
    new_fraction(mp→d_cos[k]);
  }
  number_clone(mp→half_cos[0], fraction_half_t);
  number_clone(mp→half_cos[1], twentysixbits_sqrt2_t);
  number_clone(mp→half_cos[2], zero_t);
  number_clone(mp→d_cos[0], twentyeightbits_d_t);
  number_clone(mp→d_cos[1], twentysevenbits_sqrt2_d_t);
  number_clone(mp→d_cos[2], zero_t);
  for (k = 3; k ≤ 4; k++) {
    number_clone(mp→half_cos[k], mp→half_cos[4 − k]);
    number_negate(mp→half_cos[k]);
    number_clone(mp→d_cos[k], mp→d_cos[4 − k]);
    number_negate(mp→d_cos[k]);
  }
  for (k = 5; k ≤ 7; k++) {
    number_clone(mp→half_cos[k], mp→half_cos[8 − k]);
    number_clone(mp→d_cos[k], mp→d_cos[8 − k]);
  }
```

432.   ⟨ Dealloc variables 27 ⟩ +≡
```
  for (k = 0; k ≤ 7; k++) {
    free_number(mp→half_cos[k]);
    free_number(mp→d_cos[k]);
  }
```

433.   The *convex_hull* function forces a pen polygon to be convex when it is returned by *make_pen* and after any subsequent transformation where rounding error might allow the convexity to be lost. The convex hull algorithm used here is described by F. P. Preparata and M. I. Shamos [*Computational Geometry*, Springer-Verlag, 1985].

⟨ Declarations 8 ⟩ +≡
```
  static mp_knot mp_convex_hull(MP mp, mp_knot h);
```

434.   **mp_knot** *mp_convex_hull*(**MP** *mp*, **mp_knot** *h*)
  {   /∗ Make a polygonal pen convex ∗/
    mp_knot *l*, *r*;   /∗ the leftmost and rightmost knots ∗/
    mp_knot *p*, *q*;   /∗ knots being scanned ∗/
    mp_knot *s*;   /∗ the starting point for an upcoming scan ∗/
    mp_number *dx*, *dy*;   /∗ a temporary pointer ∗/
    mp_knot *ret*;
    new_number(*dx*);
    new_number(*dy*);
    if (*pen_is_elliptical*(*h*)) {
      ret = *h*;
    }
    else {
      ⟨Set *l* to the leftmost knot in polygon *h* 435⟩;
      ⟨Set *r* to the rightmost knot in polygon *h* 436⟩;
      if (*l* ≠ *r*) {
        s = *mp_next_knot*(*r*);
        ⟨Find any knots on the path from *l* to *r* above the *l*-*r* line and move them past *r* 437⟩;
        ⟨Find any knots on the path from *s* to *l* below the *l*-*r* line and move them past *l* 441⟩;
        ⟨Sort the path from *l* to *r* by increasing *x* 442⟩;
        ⟨Sort the path from *r* to *l* by decreasing *x* 443⟩;
      }
      if (*l* ≠ *mp_next_knot*(*l*)) {
        ⟨Do a Gramm scan and remove vertices where there is no left turn 444⟩;
      }
      ret = *l*;
    }
    free_number(*dx*);
    free_number(*dy*);
    return *ret*;
  }

435.   All comparisons are done primarily on *x* and secondarily on *y*.

⟨Set *l* to the leftmost knot in polygon *h* 435⟩ ≡
  l = *h*;
  p = *mp_next_knot*(*h*);
  while (*p* ≠ *h*) {
    if (*number_lessequal*(*p*→*x_coord*, *l*→*x_coord*))
      if ((*number_less*(*p*→*x_coord*, *l*→*x_coord*)) ∨ (*number_less*(*p*→*y_coord*, *l*→*y_coord*))) *l* = *p*;
    p = *mp_next_knot*(*p*);
  }
This code is used in section 434.

436.   ⟨Set *r* to the rightmost knot in polygon *h* 436⟩ ≡
  r = *h*;
  p = *mp_next_knot*(*h*);
  while (*p* ≠ *h*) {
    if (*number_greaterequal*(*p*→*x_coord*, *r*→*x_coord*))
      if (*number_greater*(*p*→*x_coord*, *r*→*x_coord*) ∨ *number_greater*(*p*→*y_coord*, *r*→*y_coord*)) *r* = *p*;
    p = *mp_next_knot*(*p*);
  }
This code is used in section 434.

437.   ⟨Find any knots on the path from l to r above the l-r line and move them past r 437⟩ ≡

```
{
  mp_number ab_vs_cd;
  mp_number arg1, arg2;
  new_number(arg1);
  new_number(arg2);
  new_number(ab_vs_cd);
  set_number_from_substraction(dx, r→x_coord, l→x_coord);
  set_number_from_substraction(dy, r→y_coord, l→y_coord);
  p = mp_next_knot(l);
  while (p ≠ r) {
    q = mp_next_knot(p);
    set_number_from_substraction(arg1, p→y_coord, l→y_coord);
    set_number_from_substraction(arg2, p→x_coord, l→x_coord);
    ab_vs_cd(ab_vs_cd, dx, arg1, dy, arg2);
    if (number_positive(ab_vs_cd)) mp_move_knot(mp, p, r);
    p = q;
  }
  free_number(ab_vs_cd);
  free_number(arg1);
  free_number(arg2);
}
```
This code is used in section 434.

438.   The *move_knot* procedure removes p from a doubly linked list and inserts it after q.

439.   ⟨Declarations 8⟩ +≡
```
static void mp_move_knot(MP mp, mp_knot p, mp_knot q);
```

440.   void mp_move_knot(MP mp, mp_knot p, mp_knot q)
```
{
  (void) mp;
  mp_next_knot(mp_prev_knot(p)) = mp_next_knot(p);
  mp_prev_knot(mp_next_knot(p)) = mp_prev_knot(p);
  mp_prev_knot(p) = q;
  mp_next_knot(p) = mp_next_knot(q);
  mp_next_knot(q) = p;
  mp_prev_knot(mp_next_knot(p)) = p;
}
```

441. ⟨Find any knots on the path from s to l below the l-r line and move them past l 441⟩ ≡
  {
    mp_number ab_vs_cd;
    mp_number $arg1$, $arg2$;

    $new\_number(ab\_vs\_cd)$;
    $new\_number(arg1)$;
    $new\_number(arg2)$;
    $p = s$;
    while $(p \neq l)$ {
      $q = mp\_next\_knot(p)$;
      $set\_number\_from\_substraction(arg1, p\text{→}y\_coord, l\text{→}y\_coord)$;
      $set\_number\_from\_substraction(arg2, p\text{→}x\_coord, l\text{→}x\_coord)$;
      $ab\_vs\_cd(ab\_vs\_cd, dx, arg1, dy, arg2)$;
      if $(number\_negative(ab\_vs\_cd))$ $mp\_move\_knot(mp, p, l)$;
      $p = q$;
    }
    $free\_number(ab\_vs\_cd)$;
    $free\_number(arg1)$;
    $free\_number(arg2)$;
  }
This code is used in section 434.

442. The list is likely to be in order already so we just do linear insertions. Secondary comparisons on y ensure that the sort is consistent with the choice of l and r.

⟨Sort the path from l to r by increasing x 442⟩ ≡
  $p = mp\_next\_knot(l)$;
  while $(p \neq r)$ {
    $q = mp\_prev\_knot(p)$;
    while $(number\_greater(q\text{→}x\_coord, p\text{→}x\_coord))$ $q = mp\_prev\_knot(q)$;
    while $(number\_equal(q\text{→}x\_coord, p\text{→}x\_coord))$ {
      if $(number\_greater(q\text{→}y\_coord, p\text{→}y\_coord))$ $q = mp\_prev\_knot(q)$;
      else break;
    }
    if $(q \equiv mp\_prev\_knot(p))$ {
      $p = mp\_next\_knot(p)$;
    }
    else {
      $p = mp\_next\_knot(p)$;
      $mp\_move\_knot(mp, mp\_prev\_knot(p), q)$;
    }
  }
This code is used in section 434.

443.   ⟨Sort the path from r to l by decreasing x 443⟩ ≡
$p = mp\_next\_knot(r);$
while $(p \neq l)$ {
  $q = mp\_prev\_knot(p);$
  while $(number\_less(q\text{-}x\_coord, p\text{-}x\_coord))$ $q = mp\_prev\_knot(q);$
  while $(number\_equal(q\text{-}x\_coord, p\text{-}x\_coord))$ {
    if $(number\_less(q\text{-}y\_coord, p\text{-}y\_coord))$ $q = mp\_prev\_knot(q);$
    else break;
  }
  if $(q \equiv mp\_prev\_knot(p))$ {
    $p = mp\_next\_knot(p);$
  }
  else {
    $p = mp\_next\_knot(p);$
    $mp\_move\_knot(mp, mp\_prev\_knot(p), q);$
  }
}

This code is used in section 434.

444.   The condition involving *ab_vs_cd* tests if there is not a left turn at knot q. There usually will be a left turn so we streamline the case where the *then* clause is not executed.

⟨Do a Gramm scan and remove vertices where there is no left turn 444⟩ ≡
  {
    mp_number $ab\_vs\_cd;$
    mp_number $arg1,\ arg2;$

    $new\_number(arg1);$
    $new\_number(arg2);$
    $new\_number(ab\_vs\_cd);$
    $p = l;$
    $q = mp\_next\_knot(l);$
    while (1) {
      $set\_number\_from\_substraction(dx, q\text{-}x\_coord, p\text{-}x\_coord);$
      $set\_number\_from\_substraction(dy, q\text{-}y\_coord, p\text{-}y\_coord);$
      $p = q;$
      $q = mp\_next\_knot(q);$
      if $(p \equiv l)$ **break**;
      if $(p \neq r)$ {
        $set\_number\_from\_substraction(arg1, q\text{-}y\_coord, p\text{-}y\_coord);$
        $set\_number\_from\_substraction(arg2, q\text{-}x\_coord, p\text{-}x\_coord);$
        $ab\_vs\_cd(ab\_vs\_cd, dx, arg1, dy, arg2);$
        if $(number\_nonpositive(ab\_vs\_cd))$ {
          ⟨Remove knot p and back up p and q but don't go past l 445⟩;
        }
      }
    }
    $free\_number(ab\_vs\_cd);$
    $free\_number(arg1);$
    $free\_number(arg2);$
  }

This code is used in section 434.

445.   ⟨Remove knot p and back up p and q but don't go past l 445⟩ ≡

```
{
    s = mp_prev_knot(p);
    mp_xfree(p);
    mp_next_knot(s) = q;
    mp_prev_knot(q) = s;
    if (s ≡ l) {
        p = s;
    }
    else {
        p = mp_prev_knot(s);
        q = s;
    }
}
```

This code is used in section 444.

446.   The *find_offset* procedure sets global variables (cur_x, cur_y) to the offset associated with the given direction (x, y). If two different offsets apply, it chooses one of them.

static void *mp_find_offset*(**MP** *mp*, **mp_number** *x_orig*, **mp_number** *y_orig*, **mp_knot** *h*)
{
  mp_knot *p*, *q*;   /∗ consecutive knots ∗/
  if (*pen_is_elliptical*(*h*)) {
    mp_fraction xx, *yy*;   /∗ untransformed offset for an elliptical pen ∗/
    mp_number *wx*, *wy*, *hx*, *hy*;   /∗ the transformation matrix for an elliptical pen ∗/
    mp_fraction d;   /∗ a temporary register ∗/
    new_fraction(*xx*);
    new_fraction(*yy*);
    new_number(*wx*);
    new_number(*wy*);
    new_number(*hx*);
    new_number(*hy*);
    new_fraction(*d*);
    ⟨Find the offset for (x, y) on the elliptical pen *h* 450⟩ *free_number*(*xx*);
    free_number(*yy*);
    free_number(*wx*);
    free_number(*wy*);
    free_number(*hx*);
    free_number(*hy*);
    free_number(*d*);
  }
  else {
    mp_number *ab_vs_cd*;
    mp_number *arg1*, *arg2*;
    new_number(*arg1*);
    new_number(*arg2*);
    new_number(*ab_vs_cd*);
    q = *h*;
    do {
      p = *q*;
      q = *mp_next_knot*(*q*);
      set_number_from_substraction(*arg1*, *q*→*x_coord*, *p*→*x_coord*);
      set_number_from_substraction(*arg2*, *q*→*y_coord*, *p*→*y_coord*);
      ab_vs_cd(*ab_vs_cd*, *arg1*, *y_orig*, *arg2*, *x_orig*);
    } **while** (*number_negative*(*ab_vs_cd*));
    do {
      p = *q*;
      q = *mp_next_knot*(*q*);
      set_number_from_substraction(*arg1*, *q*→*x_coord*, *p*→*x_coord*);
      set_number_from_substraction(*arg2*, *q*→*y_coord*, *p*→*y_coord*);
      ab_vs_cd(*ab_vs_cd*, *arg1*, *y_orig*, *arg2*, *x_orig*);
    } **while** (*number_positive*(*ab_vs_cd*));
    number_clone(*mp*→*cur_x*, *p*→*x_coord*);
    number_clone(*mp*→*cur_y*, *p*→*y_coord*);
    free_number(*ab_vs_cd*);
    free_number(*arg1*);
    free_number(*arg2*);
  }

    }

447.   ⟨ Global variables 14 ⟩ +≡
  mp_number *cur_x*;
  mp_number *cur_y*;   /∗ all-purpose return value registers ∗/

448.   ⟨ Initialize table entries 182 ⟩ +≡
  new_number(*mp*→*cur_x*);
  new_number(*mp*→*cur_y*);

449.   ⟨ Dealloc variables 27 ⟩ +≡
  free_number(*mp*→*cur_x*);
  free_number(*mp*→*cur_y*);

450.   ⟨Find the offset for (x, y) on the elliptical pen h 450⟩ ≡
  if $(number\_zero(x\_orig) \land number\_zero(y\_orig))$ {
    $number\_clone(mp \rightarrow cur\_x, h \rightarrow x\_coord)$;
    $number\_clone(mp \rightarrow cur\_y, h \rightarrow y\_coord)$;
  }
  else {
    mp_number x, y, abs_x, abs_y;

    $new\_number(x)$;
    $new\_number(y)$;
    $new\_number(abs\_x)$;
    $new\_number(abs\_y)$;
    $number\_clone(x, x\_orig)$;
    $number\_clone(y, y\_orig)$;
    ⟨Find the non-constant part of the transformation for h 451⟩;
    $number\_clone(abs\_x, x)$;
    $number\_clone(abs\_y, y)$;
    $number\_abs(abs\_x)$;
    $number\_abs(abs\_y)$;
    while $(number\_less(abs\_x, fraction\_half\_t) \land number\_less(abs\_y, fraction\_half\_t))$ {
      $number\_double(x)$;
      $number\_double(y)$;
      $number\_clone(abs\_x, x)$;
      $number\_clone(abs\_y, y)$;
      $number\_abs(abs\_x)$;
      $number\_abs(abs\_y)$;
    }
    ⟨Make (xx, yy) the offset on the untransformed **pencircle** for the untransformed version of (x, y) 452⟩;
    {
      mp_number $r1$, $r2$;

      $new\_fraction(r1)$;
      $new\_fraction(r2)$;
      $take\_fraction(r1, xx, wx)$;
      $take\_fraction(r2, yy, hx)$;
      $number\_add(r1, r2)$;
      $set\_number\_from\_addition(mp \rightarrow cur\_x, h \rightarrow x\_coord, r1)$;
      $take\_fraction(r1, xx, wy)$;
      $take\_fraction(r2, yy, hy)$;
      $number\_add(r1, r2)$;
      $set\_number\_from\_addition(mp \rightarrow cur\_y, h \rightarrow y\_coord, r1)$;
      $free\_number(r1)$;
      $free\_number(r2)$;
    }
    $free\_number(abs\_x)$;
    $free\_number(abs\_y)$;
    $free\_number(x)$;
    $free\_number(y)$;
  }

This code is used in section 446.

451.   ⟨Find the non-constant part of the transformation for h 451⟩ ≡
  {
    $set\_number\_from\_substraction(wx, h\text{→}left\_x, h\text{→}x\_coord)$;
    $set\_number\_from\_substraction(wy, h\text{→}left\_y, h\text{→}y\_coord)$;
    $set\_number\_from\_substraction(hx, h\text{→}right\_x, h\text{→}x\_coord)$;
    $set\_number\_from\_substraction(hy, h\text{→}right\_y, h\text{→}y\_coord)$;
  }

This code is used in section 450.

452.   ⟨Make (xx, yy) the offset on the untransformed **pencircle** for the untransformed version of
      (x, y) 452⟩ ≡
  {
    mp_number $r1$, $r2$, $arg1$;

    $new\_number(arg1)$;
    $new\_fraction(r1)$;
    $new\_fraction(r2)$;
    $take\_fraction(r1, x, hy)$;
    $number\_clone(arg1, hx)$;
    $number\_negate(arg1)$;
    $take\_fraction(r2, y, arg1)$;
    $number\_add(r1, r2)$;
    $number\_negate(r1)$;
    $number\_clone(yy, r1)$;
    $number\_clone(arg1, wy)$;
    $number\_negate(arg1)$;
    $take\_fraction(r1, x, arg1)$;
    $take\_fraction(r2, y, wx)$;
    $number\_add(r1, r2)$;
    $number\_clone(xx, r1)$;
    $free\_number(arg1)$;
    $free\_number(r1)$;
    $free\_number(r2)$;
  }
  $pyth\_add(d, xx, yy)$;
  if $(number\_positive(d))$ {
    mp_number ret;

    $new\_fraction(ret)$;
    $make\_fraction(ret, xx, d)$;
    $number\_half(ret)$;
    $number\_clone(xx, ret)$;
    $make\_fraction(ret, yy, d)$;
    $number\_half(ret)$;
    $number\_clone(yy, ret)$;
    $free\_number(ret)$;
  }

This code is used in section 450.

453.   Finding the bounding box of a pen is easy except if the pen is elliptical. But we can handle that case by just calling *find_offset* twice. The answer is stored in the global variables *minx*, *maxx*, *miny*, and *maxy*.

```
static void mp_pen_bbox(MP mp, mp_knot h)
{
  mp_knot p;    /* for scanning the knot list */
  if (pen_is_elliptical(h)) {
    ⟨Find the bounding box of an elliptical pen 454⟩;
  }
  else {
    number_clone(mp_minx, h→x_coord);
    number_clone(mp_maxx, mp_minx);
    number_clone(mp_miny, h→y_coord);
    number_clone(mp_maxy, mp_miny);
    p = mp_next_knot(h);
    while (p ≠ h) {
      if (number_less(p→x_coord, mp_minx)) number_clone(mp_minx, p→x_coord);
      if (number_less(p→y_coord, mp_miny)) number_clone(mp_miny, p→y_coord);
      if (number_greater(p→x_coord, mp_maxx)) number_clone(mp_maxx, p→x_coord);
      if (number_greater(p→y_coord, mp_maxy)) number_clone(mp_maxy, p→y_coord);
      p = mp_next_knot(p);
    }
  }
}
```

454.   ⟨Find the bounding box of an elliptical pen 454⟩ ≡
```
{
  mp_number arg1, arg2;

  new_number(arg1);
  new_fraction(arg2);
  number_clone(arg2, fraction_one_t);
  mp_find_offset(mp, arg1, arg2, h);
  number_clone(mp_maxx, mp→cur_x);
  number_clone(mp_minx, h→x_coord);
  number_double(mp_minx);
  number_substract(mp_minx, mp→cur_x);
  number_negate(arg2);
  mp_find_offset(mp, arg2, arg1, h);
  number_clone(mp_maxy, mp→cur_y);
  number_clone(mp_miny, h→y_coord);
  number_double(mp_miny);
  number_substract(mp_miny, mp→cur_y);
  free_number(arg1);
  free_number(arg2);
}
```
This code is used in section 453.

455.   Numerical values.

This first set goes into the header

⟨ MPlib internal header stuff 6 ⟩ +≡

#**define** *mp_fraction* **mp_number**

#**define** *mp_angle* **mp_number**

#**define** *new_number*(A)(((**math_data** ∗)(mp→math))→allocate) (mp, &(A), mp_scaled_type)

#**define** *new_fraction*(A)(((**math_data** ∗)(mp→math))→allocate) (mp, &(A), mp_fraction_type)

#**define** *new_angle*(A)(((**math_data** ∗)(mp→math))→allocate) (mp, &(A), mp_angle_type)

#**define** *free_number*(A)(((**math_data** ∗)(mp→math))→free) (mp, &(A))

456.

#define *set_precision*()   (((**math_data** ∗)(*mp*→*math*))→*set_precision*)(*mp*)
#define *free_math*()   (((**math_data** ∗)(*mp*→*math*))→*free_math*)(*mp*)
#define *scan_numeric_token*(*A*)   (((**math_data** ∗)(*mp*→*math*))→*scan_numeric*)(*mp*, *A*)
#define *scan_fractional_token*(*A*)   (((**math_data** ∗)(*mp*→*math*))→*scan_fractional*)(*mp*, *A*)
#define *set_number_from_of_the_way*(*A*, *t*, *B*, *C*)
         (((**math_data** ∗)(*mp*→*math*))→*from_oftheway*)(*mp*, &(*A*), *t*, *B*, *C*)
#define *set_number_from_int*(*A*, *B*)   (((**math_data** ∗)(*mp*→*math*))→*from_int*)(&(*A*), *B*)
#define *set_number_from_scaled*(*A*, *B*)   (((**math_data** ∗)(*mp*→*math*))→*from_scaled*)(&(*A*), *B*)
#define *set_number_from_boolean*(*A*, *B*)   (((**math_data** ∗)(*mp*→*math*))→*from_boolean*)(&(*A*), *B*)
#define *set_number_from_double*(*A*, *B*)   (((**math_data** ∗)(*mp*→*math*))→*from_double*)(&(*A*), *B*)
#define *set_number_from_addition*(*A*, *B*, *C*)   (((**math_data** ∗)(*mp*→*math*))→*from_addition*)(&(*A*), *B*, *C*)
#define *set_number_from_substraction*(*A*, *B*, *C*)
         (((**math_data** ∗)(*mp*→*math*))→*from_substraction*)(&(*A*), *B*, *C*)
#define *set_number_from_div*(*A*, *B*, *C*)   (((**math_data** ∗)(*mp*→*math*))→*from_div*)(&(*A*), *B*, *C*)
#define *set_number_from_mul*(*A*, *B*, *C*)   (((**math_data** ∗)(*mp*→*math*))→*from_mul*)(&(*A*), *B*, *C*)
#define *number_int_div*(*A*, *C*)   (((**math_data** ∗)(*mp*→*math*))→*from_int_div*)(&(*A*), *A*, *C*)
#define *set_number_from_int_mul*(*A*, *B*, *C*)   (((**math_data** ∗)(*mp*→*math*))→*from_int_mul*)(&(*A*), *B*, *C*)

#define *set_number_to_unity*(*A*)   (((**math_data** ∗)(*mp*→*math*))→*clone*)(&(*A*), *unity_t*)
#define *set_number_to_zero*(*A*)   (((**math_data** ∗)(*mp*→*math*))→*clone*)(&(*A*), *zero_t*)
#define *set_number_to_inf*(*A*)   (((**math_data** ∗)(*mp*→*math*))→*clone*)(&(*A*), *inf_t*)
#define *set_number_to_neg_inf*(*A*)   **do**
         {
           set_number_to_inf(*A*);
           number_negate(*A*);
         }
         while (0)
#define *init_randoms*(*A*)   (((**math_data** ∗)(*mp*→*math*))→*init_randoms*)(*mp*, *A*)
#define *print_number*(*A*)   (((**math_data** ∗)(*mp*→*math*))→*print*)(*mp*, *A*)
#define *number_tostring*(*A*)   (((**math_data** ∗)(*mp*→*math*))→*tostring*)(*mp*, *A*)
#define *make_scaled*(*R*, *A*, *B*)   (((**math_data** ∗)(*mp*→*math*))→*make_scaled*)(*mp*, &(*R*), *A*, *B*)
#define *take_scaled*(*R*, *A*, *B*)   (((**math_data** ∗)(*mp*→*math*))→*take_scaled*)(*mp*, &(*R*), *A*, *B*)
#define *make_fraction*(*R*, *A*, *B*)   (((**math_data** ∗)(*mp*→*math*))→*make_fraction*)(*mp*, &(*R*), *A*, *B*)
#define *take_fraction*(*R*, *A*, *B*)   (((**math_data** ∗)(*mp*→*math*))→*take_fraction*)(*mp*, &(*R*), *A*, *B*)
#define *pyth_add*(*R*, *A*, *B*)   (((**math_data** ∗)(*mp*→*math*))→*pyth_add*)(*mp*, &(*R*), *A*, *B*)
#define *pyth_sub*(*R*, *A*, *B*)   (((**math_data** ∗)(*mp*→*math*))→*pyth_sub*)(*mp*, &(*R*), *A*, *B*)
#define *n_arg*(*R*, *A*, *B*)   (((**math_data** ∗)(*mp*→*math*))→*n_arg*)(*mp*, &(*R*), *A*, *B*)
#define *m_log*(*R*, *A*)   (((**math_data** ∗)(*mp*→*math*))→*m_log*)(*mp*, &(*R*), *A*)
#define *m_exp*(*R*, *A*)   (((**math_data** ∗)(*mp*→*math*))→*m_exp*)(*mp*, &(*R*), *A*)
#define *velocity*(*R*, *A*, *B*, *C*, *D*, *E*)   (((**math_data** ∗)(*mp*→*math*))→*velocity*)(*mp*, &(*R*), *A*, *B*, *C*, *D*, *E*)
#define *ab_vs_cd*(*R*, *A*, *B*, *C*, *D*)   (((**math_data** ∗)(*mp*→*math*))→*ab_vs_cd*)(*mp*, &(*R*), *A*, *B*, *C*, *D*)
#define *crossing_point*(*R*, *A*, *B*, *C*)   (((**math_data** ∗)(*mp*→*math*))→*crossing_point*)(*mp*, &(*R*), *A*, *B*, *C*)
#define *n_sin_cos*(*A*, *S*, *C*)   (((**math_data** ∗)(*mp*→*math*))→*sin_cos*)(*mp*, *A*, &(*S*), &(*C*))
#define *square_rt*(*A*, *S*)   (((**math_data** ∗)(*mp*→*math*))→*sqrt*)(*mp*, &(*A*), *S*)
#define *slow_add*(*R*, *A*, *B*)   (((**math_data** ∗)(*mp*→*math*))→*slow_add*)(*mp*, &(*R*), *A*, *B*)
#define *round_unscaled*(*A*)   (((**math_data** ∗)(*mp*→*math*))→*round_unscaled*)(*A*)
#define *floor_scaled*(*A*)   (((**math_data** ∗)(*mp*→*math*))→*floor_scaled*)(&(*A*))
#define *fraction_to_round_scaled*(*A*)   (((**math_data** ∗)(*mp*→*math*))→*fraction_to_round_scaled*)(&(*A*))
#define *number_to_int*(*A*)   (((**math_data** ∗)(*mp*→*math*))→*to_int*)(*A*)
#define *number_to_boolean*(*A*)   (((**math_data** ∗)(*mp*→*math*))→*to_boolean*)(*A*)
#define *number_to_scaled*(*A*)   (((**math_data** ∗)(*mp*→*math*))→*to_scaled*)(*A*)

#define *number_to_double*(*A*) (((**math_data** ∗)(*mp*→*math*))→*to_double*)(*A*)
#define *number_negate*(*A*) (((**math_data** ∗)(*mp*→*math*))→*negate*)(&(*A*))
#define *number_add*(*A*, *B*) (((**math_data** ∗)(*mp*→*math*))→*add*)(&(*A*), *B*)
#define *number_substract*(*A*, *B*) (((**math_data** ∗)(*mp*→*math*))→*substract*)(&(*A*), *B*)
#define *number_half*(*A*) (((**math_data** ∗)(*mp*→*math*))→*half*)(&(*A*))
#define *number_halfp*(*A*) (((**math_data** ∗)(*mp*→*math*))→*halfp*)(&(*A*))
#define *number_double*(*A*) (((**math_data** ∗)(*mp*→*math*))→*do_double*)(&(*A*))
#define *number_add_scaled*(*A*, *B*) (((**math_data** ∗)(*mp*→*math*))→*add_scaled*)(&(*A*), *B*)
#define *number_multiply_int*(*A*, *B*) (((**math_data** ∗)(*mp*→*math*))→*multiply_int*)(&(*A*), *B*)
#define *number_divide_int*(*A*, *B*) (((**math_data** ∗)(*mp*→*math*))→*divide_int*)(&(*A*), *B*)
#define *number_abs*(*A*) (((**math_data** ∗)(*mp*→*math*))→*abs*)(&(*A*))
#define *number_modulo*(*A*, *B*) (((**math_data** ∗)(*mp*→*math*))→*modulo*)(&(*A*), *B*)
#define *number_nonequalabs*(*A*, *B*) (((**math_data** ∗)(*mp*→*math*))→*nonequalabs*)(*A*, *B*)
#define *number_odd*(*A*) (((**math_data** ∗)(*mp*→*math*))→*odd*)(*A*)
#define *number_equal*(*A*, *B*) (((**math_data** ∗)(*mp*→*math*))→*equal*)(*A*, *B*)
#define *number_greater*(*A*, *B*) (((**math_data** ∗)(*mp*→*math*))→*greater*)(*A*, *B*)
#define *number_less*(*A*, *B*) (((**math_data** ∗)(*mp*→*math*))→*less*)(*A*, *B*)
#define *number_clone*(*A*, *B*) (((**math_data** ∗)(*mp*→*math*))→*clone*)(&(*A*), *B*)
#define *number_swap*(*A*, *B*) (((**math_data** ∗)(*mp*→*math*))→*swap*)(&(*A*), &(*B*));
#define *convert_scaled_to_angle*(*A*) (((**math_data** ∗)(*mp*→*math*))→*scaled_to_angle*)(&(*A*));
#define *convert_angle_to_scaled*(*A*) (((**math_data** ∗)(*mp*→*math*))→*angle_to_scaled*)(&(*A*));
#define *convert_fraction_to_scaled*(*A*) (((**math_data** ∗)(*mp*→*math*))→*fraction_to_scaled*)(&(*A*));
#define *convert_scaled_to_fraction*(*A*) (((**math_data** ∗)(*mp*→*math*))→*scaled_to_fraction*)(&(*A*));

#define *number_zero*(*A*) *number_equal*(*A*, *zero_t*)
#define *number_infinite*(*A*) *number_equal*(*A*, *inf_t*)
#define *number_unity*(*A*) *number_equal*(*A*, *unity_t*)
#define *number_negative*(*A*) *number_less*(*A*, *zero_t*)
#define *number_nonnegative*(*A*) (¬*number_negative*(*A*))
#define *number_positive*(*A*) *number_greater*(*A*, *zero_t*)
#define *number_nonpositive*(*A*) (¬*number_positive*(*A*))
#define *number_nonzero*(*A*) (¬*number_zero*(*A*))
#define *number_greaterequal*(*A*, *B*) (¬*number_less*(*A*, *B*))
#define *number_lessequal*(*A*, *B*) (¬*number_greater*(*A*, *B*))

457.   Edge structures.   Now we come to METAPOST's internal scheme for representing pictures. The representation is very different from METAFONT's edge structures because METAPOST pictures contain PostScript graphics objects instead of pixel images. However, the basic idea is somewhat similar in that shapes are represented via their boundaries.

The main purpose of edge structures is to keep track of graphical objects until it is time to translate them into PostScript. Since METAPOST does not need to know anything about an edge structure other than how to translate it into PostScript and how to find its bounding box, edge structures can be just linked lists of graphical objects. METAPOST has no easy way to determine whether two such objects overlap, but it suffices to draw the first one first and let the second one overwrite it if necessary.

⟨ MPlib header stuff 201 ⟩ +≡
  enum mp_graphical_object_code {
    ⟨ Graphical object codes 459 ⟩*mp_final_graphic*
  };

458.   Let's consider the types of graphical objects one at a time. First of all, a filled contour is represented by a eight-word node. The first word contains *type* and *link* fields, and the next six words contain a pointer to a cyclic path and the value to use for PostScript' **currentrgbcolor** parameter. If a pen is used for filling *pen_p*, *ljoin* and *miterlim* give the relevant information.

#define   *mp_path_p*(A)   (A)→*path_p_*   /∗ a pointer to the path that needs filling ∗/
#define   *mp_pen_p*(A)   (A)→*pen_p_*   /∗ a pointer to the pen to fill or stroke with ∗/
#define   **mp_color_model**(A)   ((*mp_fill_node*)(A))→*color_model_*   /∗ the color model ∗/
#define   *cyan*   *red*
#define   *grey*   *red*
#define   *magenta*   *green*
#define   *yellow*   *blue*
#define   *mp_pre_script*(A)   ((*mp_fill_node*)(A))→*pre_script_*
#define   *mp_post_script*(A)   ((*mp_fill_node*)(A))→*post_script_*
⟨ MPlib internal header stuff 6 ⟩ +≡
  typedef struct mp_fill_node_data {
    NODE_BODY;

    halfword *color_model_*;
    mp_number *red*;
    mp_number *green*;
    mp_number *blue*;
    mp_number *black*;
    mp_string *pre_script_*;
    mp_string *post_script_*;
    mp_knot *path_p_*;
    mp_knot *pen_p_*;
    unsigned char *ljoin*;
    mp_number *miterlim*;
  } **mp_fill_node_data**;
  typedef struct mp_fill_node_data ∗**mp_fill_node**;

459.   ⟨ Graphical object codes 459 ⟩ ≡
  mp_fill_code = 1 ,

See also sections 463, 470, 474, and 1267.

This code is used in section 457.

460.   Make a fill node for cyclic path p and color black.

#define $fill\_node\_size$ **sizeof**(**struct mp_fill_node_data**)

  static mp_node $mp\_new\_fill\_node$(**MP** mp, **mp_knot** p)
  {
    mp_fill_node $t = malloc\_node(fill\_node\_size)$;
    $mp\_type(t) = mp\_fill\_node\_type$;
    $mp\_path\_p(t) = p$;
    $mp\_pen\_p(t) = \Lambda$;   /∗ Λ means don't use a pen ∗/
    $new\_number(t\rightarrow red)$;
    $new\_number(t\rightarrow green)$;
    $new\_number(t\rightarrow blue)$;
    $new\_number(t\rightarrow black)$;
    $new\_number(t\rightarrow miterlim)$;
    $clear\_color(t)$;
    mp_color_model$(t) = mp\_uninitialized\_model$;
    $mp\_pre\_script(t) = \Lambda$;
    $mp\_post\_script(t) = \Lambda$;   /∗ Set the $ljoin$ and $miterlim$ fields in object t ∗/
    if $(number\_greater(internal\_value(mp\_linejoin), unity\_t))$ $t\rightarrow ljoin = 2$;
    else if $(number\_positive(internal\_value(mp\_linejoin)))$ $t\rightarrow ljoin = 1$;
    else $t\rightarrow ljoin = 0$;
    if $(number\_less(internal\_value(mp\_miterlimit), unity\_t))$ {
      $set\_number\_to\_unity(t\rightarrow miterlim)$;
    }
    else {
      $number\_clone(t\rightarrow miterlim, internal\_value(mp\_miterlimit))$;
    }
    return (**mp_node**) t;
  }

461.   **static void** $mp\_free\_fill\_node$(**MP** mp, **mp_fill_node** p)
  {
    $mp\_toss\_knot\_list(mp, mp\_path\_p(p))$;
    if $(mp\_pen\_p(p) \neq \Lambda)$ $mp\_toss\_knot\_list(mp, mp\_pen\_p(p))$;
    if $(mp\_pre\_script(p) \neq \Lambda)$ $delete\_str\_ref(mp\_pre\_script(p))$;
    if $(mp\_post\_script(p) \neq \Lambda)$ $delete\_str\_ref(mp\_post\_script(p))$;
    $free\_number(p\rightarrow red)$;
    $free\_number(p\rightarrow green)$;
    $free\_number(p\rightarrow blue)$;
    $free\_number(p\rightarrow black)$;
    $free\_number(p\rightarrow miterlim)$;
    $mp\_free\_node(mp, (\textbf{mp\_node})\ p, fill\_node\_size)$;
  }

462.   A stroked path is represented by an eight-word node that is like a filled contour node except that it contains the current **linecap** value, a scale factor for the dash pattern, and a pointer that is non-NULL if the stroke is to be dashed. The purpose of the scale factor is to allow a picture to be transformed without touching the picture that *dash_p* points to.

#define   *mp_dash_p*(*A*)   ((*mp_stroked_node*)(*A*))→*dash_p_*
            /∗ a pointer to the edge structure that gives the dash pattern ∗/

⟨ MPlib internal header stuff 6 ⟩ +≡
  typedef struct mp_stroked_node_data {
    NODE_BODY;

    halfword *color_model_*;
    mp_number *red*;
    mp_number *green*;
    mp_number *blue*;
    mp_number *black*;
    mp_string *pre_script_*;
    mp_string *post_script_*;
    mp_knot *path_p_*;
    mp_knot *pen_p_*;
    unsigned char *ljoin*;
    mp_number *miterlim*;
    unsigned char *lcap*;
    mp_node *dash_p_*;
    mp_number *dash_scale*;
  } **mp_stroked_node_data**;
  typedef struct mp_stroked_node_data ∗**mp_stroked_node**;

463.   ⟨ Graphical object codes 459 ⟩ +≡
  mp_stroked_code = 2 ,

464.   Make a stroked node for path p with $mp\_pen\_p(p)$ temporarily Λ.

#define $stroked\_node\_size$ **sizeof**(**struct mp_stroked_node_data**)

  static mp_node $mp\_new\_stroked\_node(\textbf{MP } mp, \textbf{mp\_knot } p)$
  {
    mp_stroked_node $t = malloc\_node(stroked\_node\_size)$;
    $mp\_type(t) = mp\_stroked\_node\_type$;
    $mp\_path\_p(t) = p$;
    $mp\_pen\_p(t) = \Lambda$;
    $mp\_dash\_p(t) = \Lambda$;
    $new\_number(t\text{-}dash\_scale)$;
    $set\_number\_to\_unity(t\text{-}dash\_scale)$;
    $new\_number(t\text{-}red)$;
    $new\_number(t\text{-}green)$;
    $new\_number(t\text{-}blue)$;
    $new\_number(t\text{-}black)$;
    $new\_number(t\text{-}miterlim)$;
    $clear\_color(t)$;
    $mp\_pre\_script(t) = \Lambda$;
    $mp\_post\_script(t) = \Lambda$;   /∗ Set the $ljoin$ and $miterlim$ fields in object t ∗/
    if $(number\_greater(internal\_value(mp\_linejoin), unity\_t))$ $t\text{-}ljoin = 2$;
    else if $(number\_positive(internal\_value(mp\_linejoin)))$ $t\text{-}ljoin = 1$;
    else $t\text{-}ljoin = 0$;
    if $(number\_less(internal\_value(mp\_miterlimit), unity\_t))$ {
      $set\_number\_to\_unity(t\text{-}miterlim)$;
    }
    else {
      $number\_clone(t\text{-}miterlim, internal\_value(mp\_miterlimit))$;
    }
    if $(number\_greater(internal\_value(mp\_linecap), unity\_t))$ $t\text{-}lcap = 2$;
    else if $(number\_positive(internal\_value(mp\_linecap)))$ $t\text{-}lcap = 1$;
    else $t\text{-}lcap = 0$;
    return (**mp_node**) t;
  }

465.   **static** $mp\_edge\_header\_node$ $mp\_free\_stroked\_node(\textbf{MP } mp, \textbf{mp\_stroked\_node } p)$
  {
    $mp\_edge\_header\_node\, e = \Lambda$;
    $mp\_toss\_knot\_list(mp, mp\_path\_p(p))$;
    if $(mp\_pen\_p(p) \neq \Lambda)$ $mp\_toss\_knot\_list(mp, mp\_pen\_p(p))$;
    if $(mp\_pre\_script(p) \neq \Lambda)$ $delete\_str\_ref(mp\_pre\_script(p))$;
    if $(mp\_post\_script(p) \neq \Lambda)$ $delete\_str\_ref(mp\_post\_script(p))$;
    $e = (mp\_edge\_header\_node)mp\_dash\_p(p)$;
    $free\_number(p\text{-}dash\_scale)$;
    $free\_number(p\text{-}red)$;
    $free\_number(p\text{-}green)$;
    $free\_number(p\text{-}blue)$;
    $free\_number(p\text{-}black)$;
    $free\_number(p\text{-}miterlim)$;
    $mp\_free\_node(mp, (\textbf{mp\_node}) p, stroked\_node\_size)$;
    return e;
  }

466.   When a dashed line is computed in a transformed coordinate system, the dash lengths get scaled like the pen shape and we need to compensate for this. Since there is no unique scale factor for an arbitrary transformation, we use the the square root of the determinant. The properties of the determinant make it easier to maintain the *dash_scale*. The computation is fairly straight-forward except for the initialization of the scale factor *s*. The factor of 64 is needed because *square_rt* scales its result by 2^8 while we need 2^{14} to counteract the effect of *take_fraction*.

467.   **void** *mp_sqrt_det*(**MP** *mp*, **mp_number** *∗ret*, **mp_number** *a_orig*, **mp_number**
          b_orig, **mp_number** *c_orig*, **mp_number** *d_orig*)
{
    mp_number *a*, *b*, *c*, *d*;
    mp_number *maxabs*;   /∗ *max*(*a*, *b*, *c*, *d*) ∗/
    unsigned *s*;   /∗ amount by which the result of *square_rt* needs to be scaled ∗/
    new_number(*a*);
    new_number(*b*);
    new_number(*c*);
    new_number(*d*);
    new_number(*maxabs*);
    number_clone(*a*, *a_orig*);
    number_clone(*b*, *b_orig*);
    number_clone(*c*, *c_orig*);
    number_clone(*d*, *d_orig*);   /∗ Initialize *maxabs* ∗/
    {
        mp_number *tmp*;

        new_number(*tmp*);
        number_clone(*maxabs*, *a*);
        number_abs(*maxabs*);
        number_clone(*tmp*, *b*);
        number_abs(*tmp*);
        if (*number_greater*(*tmp*, *maxabs*)) *number_clone*(*maxabs*, *tmp*);
        number_clone(*tmp*, *c*);
        number_abs(*tmp*);
        if (*number_greater*(*tmp*, *maxabs*)) *number_clone*(*maxabs*, *tmp*);
        number_clone(*tmp*, *d*);
        number_abs(*tmp*);
        if (*number_greater*(*tmp*, *maxabs*)) *number_clone*(*maxabs*, *tmp*);
        free_number(*tmp*);
    }
    s = 64;
    while ((*number_less*(*maxabs*, *fraction_one_t*)) ∧ (*s* > 1)) {
        number_double(*a*);
        number_double(*b*);
        number_double(*c*);
        number_double(*d*);
        number_double(*maxabs*);
        s = *s*/2;
    }
    {
        mp_number *r1*, *r2*;

        new_fraction(*r1*);
        new_fraction(*r2*);
        take_fraction(*r1*, *a*, *d*);
        take_fraction(*r2*, *b*, *c*);
        number_substract(*r1*, *r2*);
        number_abs(*r1*);
        square_rt(*∗ret*, *r1*);
        number_multiply_int(*∗ret*, *s*);
        free_number(*r1*);

```
        free_number (r2 );
    }
    free_number (a);
    free_number (b);
    free_number (c);
    free_number (d);
    free_number (maxabs );
}
static void mp_get_pen_scale (MP mp, mp_number *ret, mp_knot p)
{
    if (p ≡ Λ) {
        set_number_to_zero (*ret );
    }
    else {
        mp_number a, b, c, d;

        new_number (a);
        new_number (b);
        new_number (c);
        new_number (d);
        set_number_from_substraction (a, p→left_x , p→x_coord );
        set_number_from_substraction (b, p→right_x , p→x_coord );
        set_number_from_substraction (c, p→left_y , p→y_coord );
        set_number_from_substraction (d, p→right_y , p→y_coord );
        mp_sqrt_det (mp, ret , a, b, c, d);
        free_number (a);
        free_number (b);
        free_number (c);
        free_number (d);
    }
}
```

468.   ⟨ Declarations 8 ⟩ +≡
    static void mp_sqrt_det (**MP** mp, **mp_number** *ret, **mp_number** a, **mp_number** b, **mp_number**
        c, **mp_number** d);

469.   When a picture contains text, this is represented by a fourteen-word node where the color information and *type* and *link* fields are augmented by additional fields that describe the text and how it is transformed. The *path_p* and *mp_pen_p* pointers are replaced by a number that identifies the font and a string number that gives the text to be displayed. The *width*, *height*, and *depth* fields give the dimensions of the text at its design size, and the remaining six words give a transformation to be applied to the text. The *new_text_node* function initializes everything to default values so that the text comes out black with its reference point at the origin.

#define   *mp_text_p*(*A*)   ((*mp_text_node*)(*A*))→*text_p_*   /* a string pointer for the text to display */
#define   *mp_font_n*(*A*)   ((*mp_text_node*)(*A*))→*font_n_*   /* the font number */

⟨MPlib internal header stuff 6⟩ +≡
  typedef struct mp_text_node_data {
    NODE_BODY;

    halfword *color_model_*;
    mp_number *red*;
    mp_number *green*;
    mp_number *blue*;
    mp_number *black*;
    mp_string *pre_script_*;
    mp_string *post_script_*;
    mp_string *text_p_*;
    halfword *font_n_*;
    mp_number *width*;
    mp_number *height*;
    mp_number *depth*;
    mp_number *tx*;
    mp_number *ty*;
    mp_number *txx*;
    mp_number *txy*;
    mp_number *tyx*;
    mp_number *tyy*;
  } **mp_text_node_data**;
  typedef struct mp_text_node_data ∗mp_text_node;

470.   ⟨Graphical object codes 459⟩ +≡
  mp_text_code = 3 ,

471.   Make a text node for font f and text string s.

#define *text_node_size*   **sizeof**(**struct mp_text_node_data**)

  static mp_node *mp_new_text_node*(**MP** *mp*, **char** *∗f*, **mp_string** *s*)
  {
    mp_text_node *t* = *malloc_node*(*text_node_size*);
    mp_type(*t*) = *mp_text_node_type*;
    mp_text_p(*t*) = *s*;
    add_str_ref(*s*);
    mp_font_n(*t*) = (**halfword**) *mp_find_font*(*mp*, *f*);   /∗ this identifies the font ∗/
    new_number(*t→red*);
    new_number(*t→green*);
    new_number(*t→blue*);
    new_number(*t→black*);
    new_number(*t→width*);
    new_number(*t→height*);
    new_number(*t→depth*);
    clear_color(*t*);
    mp_pre_script(*t*) = Λ;
    mp_post_script(*t*) = Λ;
    new_number(*t→tx*);
    new_number(*t→ty*);
    new_number(*t→txx*);
    new_number(*t→txy*);
    new_number(*t→tyx*);
    new_number(*t→tyy*);   /∗ *tx_val*(*t*) = 0; *ty_val*(*t*) = 0; ∗/   /∗ *txy_val*(*t*) = 0; *tyx_val*(*t*) = 0; ∗/
    set_number_to_unity(*t→txx*);
    set_number_to_unity(*t→tyy*);
    mp_set_text_box(*mp*, *t*);   /∗ this finds the bounding box ∗/
    return (**mp_node**) *t*;
  }

472.   **static void** *mp_free_text_node*(**MP** *mp*, **mp_text_node** *p*)
  {   /∗ *delete_str_ref*(*mp_text_p*(*p*)); ∗/   /∗ gives errors ∗/
  if (*mp_pre_script*(*p*) ≠ Λ) *delete_str_ref*(*mp_pre_script*(*p*));
  if (*mp_post_script*(*p*) ≠ Λ) *delete_str_ref*(*mp_post_script*(*p*));
  free_number(*p→red*);
  free_number(*p→green*);
  free_number(*p→blue*);
  free_number(*p→black*);
  free_number(*p→width*);
  free_number(*p→height*);
  free_number(*p→depth*);
  free_number(*p→tx*);
  free_number(*p→ty*);
  free_number(*p→txx*);
  free_number(*p→txy*);
  free_number(*p→tyx*);
  free_number(*p→tyy*);
  mp_free_node(*mp*, (**mp_node**) *p*, *text_node_size*);
  }

473.   The last two types of graphical objects that can occur in an edge structure are clipping paths and **setbounds** paths. These are slightly more difficult to implement because we must keep track of exactly what is being clipped or bounded when pictures get merged together. For this reason, each clipping or **setbounds** operation is represented by a pair of nodes: first comes a node whose *path_p* gives the relevant path, then there is the list of objects to clip or bound followed by a closing node.

#define   *has_color*(A)   (*mp_type*((A)) < *mp_start_clip_node_type*)
      /∗ does a graphical object have color fields? ∗/
#define   *has_pen*(A)   (*mp_type*((A)) < *mp_text_node_type*)
      /∗ does a graphical object have a *mp_pen_p* field? ∗/
#define   *is_start_or_stop*(A)   (*mp_type*((A)) ≥ *mp_start_clip_node_type*)
#define   *is_stop*(A)   (*mp_type*((A)) ≥ *mp_stop_clip_node_type*)

⟨MPlib internal header stuff 6⟩ +≡
  typedef struct mp_start_clip_node_data {
    NODE_BODY;

    mp_knot *path_p_*;
  } **mp_start_clip_node_data**;
  typedef struct mp_start_clip_node_data ∗**mp_start_clip_node**;
  typedef struct mp_start_bounds_node_data {
    NODE_BODY;

    mp_knot *path_p_*;
  } **mp_start_bounds_node_data**;
  typedef struct mp_start_bounds_node_data ∗**mp_start_bounds_node**;
  typedef struct mp_stop_clip_node_data {
    NODE_BODY;
  } **mp_stop_clip_node_data**;
  typedef struct mp_stop_clip_node_data ∗**mp_stop_clip_node**;
  typedef struct mp_stop_bounds_node_data {
    NODE_BODY;
  } **mp_stop_bounds_node_data**;
  typedef struct mp_stop_bounds_node_data ∗**mp_stop_bounds_node**;

474.   ⟨Graphical object codes 459⟩ +≡
  mp_start_clip_code = 4,   /∗ *type* of a node that starts clipping ∗/
  mp_start_bounds_code = 5,   /∗ *type* of a node that gives a **setbounds** path ∗/
  mp_stop_clip_code = 6,   /∗ *type* of a node that stops clipping ∗/
  mp_stop_bounds_code = 7 ,   /∗ *type* of a node that stops **setbounds** ∗/

475.

```
#define start_clip_size   sizeof (struct mp_start_clip_node_data)
#define stop_clip_size    sizeof (struct mp_stop_clip_node_data)
#define start_bounds_size  sizeof (struct mp_start_bounds_node_data)
#define stop_bounds_size   sizeof (struct mp_stop_bounds_node_data)
  static mp_node mp_new_bounds_node (MP mp, mp_knot p, quarterword c)
  {    /* make a node of type c where p is the clipping or setbounds path */
    if (c ≡ mp_start_clip_node_type) {
      mp_start_clip_node t;    /* the new node */

      t = (mp_start_clip_node) malloc_node (start_clip_size);
      t⃗path_p_ = p;
      mp_type (t) = c;
      t⃗link = Λ;
      return (mp_node) t;
    }
    else if (c ≡ mp_start_bounds_node_type) {
      mp_start_bounds_node t;    /* the new node */

      t = (mp_start_bounds_node) malloc_node (start_bounds_size);
      t⃗path_p_ = p;
      mp_type (t) = c;
      t⃗link = Λ;
      return (mp_node) t;
    }
    else if (c ≡ mp_stop_clip_node_type) {
      mp_stop_clip_node t;    /* the new node */

      t = (mp_stop_clip_node) malloc_node (stop_clip_size);
      mp_type (t) = c;
      t⃗link = Λ;
      return (mp_node) t;
    }
    else if (c ≡ mp_stop_bounds_node_type) {
      mp_stop_bounds_node t;    /* the new node */

      t = (mp_stop_bounds_node) malloc_node (stop_bounds_size);
      mp_type (t) = c;
      t⃗link = Λ;
      return (mp_node) t;
    }
    else {
      assert (0);
    }
    return Λ;
  }
```

476.   **static void** *mp_free_start_clip_node*(**MP** *mp*, **mp_start_clip_node** *p*)
{
  mp_toss_knot_list(*mp*, *mp_path_p*(*p*));
  mp_free_node(*mp*, (**mp_node**) *p*, *start_clip_size*);
}
static void *mp_free_start_bounds_node*(**MP** *mp*, **mp_start_bounds_node** *p*)
{
  mp_toss_knot_list(*mp*, *mp_path_p*(*p*));
  mp_free_node(*mp*, (**mp_node**) *p*, *start_bounds_size*);
}
static void *mp_free_stop_clip_node*(**MP** *mp*, **mp_stop_clip_node** *p*)
{
  mp_free_node(*mp*, (**mp_node**) *p*, *stop_clip_size*);
}
static void *mp_free_stop_bounds_node*(**MP** *mp*, **mp_stop_bounds_node** *p*)
{
  mp_free_node(*mp*, (**mp_node**) *p*, *stop_bounds_size*);
}

477.   All the essential information in an edge structure is encoded as a linked list of graphical objects as we have just seen, but it is helpful to add some redundant information. A single edge structure might be used as a dash pattern many times, and it would be nice to avoid scanning the same structure repeatedly. Thus, an edge structure known to be a suitable dash pattern has a header that gives a list of dashes in a sorted order designed for rapid translation into PostScript.

Each dash is represented by a three-word node containing the initial and final x coordinates as well as the usual *link* field. The *link* fields points to the dash node with the next higher x-coordinates and the final link points to a special location called *null_dash*. (There should be no overlap between dashes). Since the y coordinate of the dash pattern is needed to determine the period of repetition, this needs to be stored in the edge header along with a pointer to the list of dash nodes.

The *dash_info* is explained below.

#define *dash_list*(*A*) (*mp_dash_node*)(((*mp_dash_node*)(*A*))⇁*link*)
           /∗ in an edge header this points to the first dash node ∗/
#define *set_dash_list*(*A*, *B*) ((*mp_dash_node*)(*A*))⇁*link* = (**mp_node**)((*B*))
           /∗ in an edge header this points to the first dash node ∗/
⟨MPlib internal header stuff 6⟩ +≡
  typedef struct mp_dash_node_data {
    NODE_BODY;

    mp_number *start_x*;   /∗ the starting x coordinate in a dash node ∗/
    mp_number *stop_x*;   /∗ the ending x coordinate in a dash node ∗/
    mp_number *dash_y*;   /∗ y value for the dash list in an edge header ∗/
    mp_node *dash_info_*;
  } **mp_dash_node_data**;

478.   ⟨Types in the outer block 33⟩ +≡
  typedef struct mp_dash_node_data ∗**mp_dash_node**;

479.   ⟨Initialize table entries 182⟩ +≡
  mp⇁*null_dash* = *mp_get_dash_node*(*mp*);

480.   ⟨Free table entries 183⟩ +≡
  mp_free_node(*mp*, (**mp_node**) *mp*⇁*null_dash*, *dash_node_size*);

481.

#define *dash_node_size*   **sizeof**(**struct mp_dash_node_data**)

  static mp_dash_node *mp_get_dash_node*(**MP** *mp*)

  {

    mp_dash_node *p* = (**mp_dash_node**) *malloc_node*(*dash_node_size*);

    p⇀has_number = 0;

    new_number(*p⇀start_x*);

    new_number(*p⇀stop_x*);

    new_number(*p⇀dash_y*);

    mp_type(*p*) = *mp_dash_node_type*;

    return *p*;

  }

482.   It is also convenient for an edge header to contain the bounding box information needed by the **llcorner** and **urcorner** operators so that this does not have to be recomputed unnecessarily. This is done by adding fields for the x and y extremes as well as a pointer that indicates how far the bounding box computation has gotten. Thus if the user asks for the bounding box and then adds some more text to the picture before asking for more bounding box information, the second computation need only look at the additional text.

When the bounding box has not been computed, the *bblast* pointer points to a dummy link at the head of the graphical object list while the *minx_val* and *miny_val* fields contain `EL_GORDO` and the *maxx_val* and *maxy_val* fields contain $-$`EL_GORDO`.

Since the bounding box of pictures containing objects of type **mp_start_bounds_node** depends on the value of **truecorners**, the bounding box data might not be valid for all values of this parameter. Hence, the *bbtype* field is needed to keep track of this.

#define *bblast*(*A*)   ((*mp_edge_header_node*)(*A*))⇀*bblast_*

      /∗ last item considered in bounding box computation ∗/

#define *edge_list*(*A*)   ((*mp_edge_header_node*)(*A*))⇀*list_*

      /∗ where the object list begins in an edge header ∗/

⟨ MPlib internal header stuff 6 ⟩ +≡

  typedef struct mp_edge_header_node_data {

    `NODE_BODY`;

    mp_number *start_x*;

    mp_number *stop_x*;

    mp_number *dash_y*;

    mp_node *dash_info_*;

    mp_number *minx*;

    mp_number *miny*;

    mp_number *maxx*;

    mp_number *maxy*;

    mp_node *bblast_*;

    int *bbtype*;   /∗ tells how bounding box data depends on **truecorners** ∗/

    mp_node *list_*;

    mp_node *obj_tail_*;   /∗ explained below ∗/

    halfword *ref_count_*;   /∗ explained below ∗/

  } **mp_edge_header_node_data**;

  typedef struct mp_edge_header_node_data *∗***mp_edge_header_node**;

483.

#define *no_bounds* 0   /∗ *bbtype* value when bounding box data is valid for all **truecorners** values ∗/
#define *bounds_set* 1   /∗ *bbtype* value when bounding box data is for **truecorners** ≤ 0 ∗/
#define *bounds_unset* 2   /∗ *bbtype* value when bounding box data is for **truecorners** > 0 ∗/

 static void *mp_init_bbox*(**MP** *mp*, **mp_edge_header_node** *h*)
 {   /∗ Initialize the bounding box information in edge structure *h* ∗/
  (**void**) *mp*;
  bblast(*h*) = *edge_list*(*h*);
  h⇀bbtype = *no_bounds*;
  set_number_to_inf(*h⇀minx*);
  set_number_to_inf(*h⇀miny*);
  set_number_to_neg_inf(*h⇀maxx*);
  set_number_to_neg_inf(*h⇀maxy*);
 }

484.   The only other entries in an edge header are a reference count in the first word and a pointer to the tail of the object list in the last word.

#define *obj_tail*(*A*) ((**mp_edge_header_node**)(*A*))⇀*obj_tail_*
   /∗ points to the last entry in the object list ∗/
#define *edge_ref_count*(*A*) ((**mp_edge_header_node**)(*A*))⇀*ref_count_*
#define *edge_header_size* **sizeof**(**struct mp_edge_header_node_data**)

 static mp_edge_header_node *mp_get_edge_header_node*(**MP** *mp*)
 {
  mp_edge_header_node *p* = (**mp_edge_header_node**) *malloc_node*(*edge_header_size*);

  mp_type(*p*) = *mp_edge_header_node_type*;
  new_number(*p⇀start_x*);
  new_number(*p⇀stop_x*);
  new_number(*p⇀dash_y*);
  new_number(*p⇀minx*);
  new_number(*p⇀miny*);
  new_number(*p⇀maxx*);
  new_number(*p⇀maxy*);
  *p⇀list_* = *mp_get_token_node*(*mp*);   /∗ or whatever, just a need a link handle ∗/
  return *p*;
 }
 static void *mp_init_edges*(**MP** *mp*, **mp_edge_header_node** *h*)
 {   /∗ initialize an edge header to NULL values ∗/
  set_dash_list(*h*, *mp⇀null_dash*);
  obj_tail(*h*) = *edge_list*(*h*);
  mp_link(*edge_list*(*h*)) = Λ;
  edge_ref_count(*h*) = 0;
  mp_init_bbox(*mp*, *h*);
 }

485.   Here is how edge structures are deleted. The process can be recursive because of the need to dereference edge structures that are used as dash patterns.

#define $add\_edge\_ref\,(A)$ $incr\,(edge\_ref\_count\,((A)))$
#define $delete\_edge\_ref\,(A)$
       {
        if $(edge\_ref\_count\,((A)) \equiv 0)$ $mp\_toss\_edges\,(mp, (\textbf{mp\_edge\_header\_node})(A));$
        else $decr\,(edge\_ref\_count\,((A)));$
       }

⟨ Declarations 8 ⟩ +≡
    static void $mp\_flush\_dash\_list\,(\textbf{MP}\;\;mp, \textbf{mp\_edge\_header\_node}\;\;h);$
    static mp_edge_header_node $mp\_toss\_gr\_object\,(\textbf{MP}\;\;mp, \textbf{mp\_node}\;\;p);$
    static void $mp\_toss\_edges\,(\textbf{MP}\;\;mp, \textbf{mp\_edge\_header\_node}\;\;h);$

486.   **void** $mp\_toss\_edges(\textbf{MP}\ mp, \textbf{mp\_edge\_header\_node}\ h)$
{
    mp_node p, q;   /∗ pointers that scan the list being recycled ∗/
    mp_edge_header_node r;   /∗ an edge structure that object p refers to ∗/
    $mp\_flush\_dash\_list(mp, h);$
    $q = mp\_link(edge\_list(h));$
    while $((q \neq \Lambda))$ {
       $p = q;$
       $q = mp\_link(q);$
       $r = mp\_toss\_gr\_object(mp, p);$
       if $(r \neq \Lambda)$ $delete\_edge\_ref(r);$
    }
    $free\_number(h{\to}start\_x);$
    $free\_number(h{\to}stop\_x);$
    $free\_number(h{\to}dash\_y);$
    $free\_number(h{\to}minx);$
    $free\_number(h{\to}miny);$
    $free\_number(h{\to}maxx);$
    $free\_number(h{\to}maxy);$
    $mp\_free\_token\_node(mp, h{\to}list\_);$
    $mp\_free\_node(mp, (\textbf{mp\_node})\ h, edge\_header\_size);$
}
void $mp\_flush\_dash\_list(\textbf{MP}\ mp, \textbf{mp\_edge\_header\_node}\ h)$
{
    mp_dash_node p, q;   /∗ pointers that scan the list being recycled ∗/
    $q = dash\_list(h);$
    while $(q \neq mp{\to}null\_dash)$ {   /∗ todo: NULL check should not be needed ∗/
       $p = q;$
       $q = (\textbf{mp\_dash\_node})\ mp\_link(q);$
       $mp\_free\_node(mp, (\textbf{mp\_node})\ p, dash\_node\_size);$
    }
    $set\_dash\_list(h, mp{\to}null\_dash);$
}
mp_edge_header_node $mp\_toss\_gr\_object(\textbf{MP}\ mp, \textbf{mp\_node}\ p)$
{   /∗ returns an edge structure that needs to be dereferenced ∗/
    mp_edge_header_node $e = \Lambda;$ /∗ the edge structure to return ∗/
    switch $(mp\_type(p))$ {
    case $mp\_fill\_node\_type$: $mp\_free\_fill\_node(mp, (\textbf{mp\_fill\_node})\ p);$
       break;
    case $mp\_stroked\_node\_type$: $e = mp\_free\_stroked\_node(mp, (\textbf{mp\_stroked\_node})\ p);$
       break;
    case $mp\_text\_node\_type$: $mp\_free\_text\_node(mp, (\textbf{mp\_text\_node})\ p);$
       break;
    case $mp\_start\_clip\_node\_type$: $mp\_free\_start\_clip\_node(mp, (\textbf{mp\_start\_clip\_node})\ p);$
       break;
    case $mp\_start\_bounds\_node\_type$: $mp\_free\_start\_bounds\_node(mp, (\textbf{mp\_start\_bounds\_node})\ p);$
       break;
    case $mp\_stop\_clip\_node\_type$: $mp\_free\_stop\_clip\_node(mp, (\textbf{mp\_stop\_clip\_node})\ p);$
       break;
    case $mp\_stop\_bounds\_node\_type$: $mp\_free\_stop\_bounds\_node(mp, (\textbf{mp\_stop\_bounds\_node})\ p);$
       break;

```
      default:       /* there are no other valid cases, but please the compiler */
        break;
      }
      return e;
    }
```

487.   If we use *add_edge_ref* to "copy" edge structures, the real copying needs to be done before making a significant change to an edge structure. Much of the work is done in a separate routine *copy_objects* that copies a list of graphical objects into a new edge header.

```
    static mp_edge_header_node mp_private_edges(MP mp, mp_edge_header_node h)
    {      /* make a private copy of the edge structure headed by h */
      mp_edge_header_node hh;        /* the edge header for the new copy */
      mp_dash_node p, pp;       /* pointers for copying the dash list */

      assert(mp_type(h) ≡ mp_edge_header_node_type);
      if (edge_ref_count(h) ≡ 0) {
        return h;
      }
      else {
        decr(edge_ref_count(h));
        hh = (mp_edge_header_node) mp_copy_objects(mp, mp_link(edge_list(h)), Λ);
        ⟨Copy the dash list from h to hh 488⟩;
        ⟨Copy the bounding box information from h to hh and make bblast(hh) point into the new object
            list 490⟩;
        return hh;
      }
    }
```

488.   Here we use the fact that $dash\_list(hh) = mp\_link(hh)$.

⟨Copy the dash list from *h* to *hh* 488⟩ ≡

```
  pp = (mp_dash_node) hh;
  p = dash_list(h);
  while ((p ≠ mp→null_dash)) {
    mp_link(pp) = (mp_node) mp_get_dash_node(mp);
    pp = (mp_dash_node) mp_link(pp);
    number_clone(pp→start_x, p→start_x);
    number_clone(pp→stop_x, p→stop_x);
    p = (mp_dash_node) mp_link(p);
  }
  mp_link(pp) = (mp_node) mp→null_dash; number_clone(hh→dash_y, h→dash_y)
```

This code is used in section 487.

489.   *h* is an edge structure

```
static mp_dash_object *mp_export_dashes(MP mp, mp_stroked_node q, mp_number w)
{
  mp_dash_object * d;

  mp_dash_node p, h;
  mp_number scf ;      /* scale factor */
  mp_number dashoff ;
  double *dashes = Λ;
  int num_dashes = 1;

  h = (mp_dash_node) mp_dash_p(q);
  if (h ≡ Λ ∨ dash_list(h) ≡ mp→null_dash ) return Λ;
  new_number (scf );
  p = dash_list(h);
  mp_get_pen_scale(mp, &scf , mp_pen_p(q));
  if (number_zero(scf )) {
    if (number_zero(w)) {
      number_clone(scf , q→dash_scale);
    }
    else {
      free_number (scf );
      return Λ;
    }
  }
  else {
    mp_number ret;

    new_number (ret);
    make_scaled (ret, w, scf );
    take_scaled (scf , ret, q→dash_scale);
    free_number (ret);
  }
  number_clone(w, scf );
  d = xmalloc(1, sizeof (mp_dash_object));
  add_var_used (sizeof (mp_dash_object));
  set_number_from_addition (mp→null_dash→start_x , p→start_x , h→dash_y );
  {
    mp_number ret, arg1 ;

    new_number (ret);
    new_number (arg1 );
    new_number (dashoff );
    while (p ≠ mp→null_dash ) {
      dashes = xrealloc(dashes, (num_dashes + 2), sizeof (double));
      set_number_from_substraction (arg1 , p→stop_x , p→start_x );
      take_scaled (ret, arg1 , scf );
      dashes[(num_dashes − 1)] = number_to_double(ret);
      set_number_from_substraction (arg1 , ((mp_dash_node) mp_link (p))→start_x , p→stop_x );
      take_scaled (ret, arg1 , scf );
      dashes[(num_dashes)] = number_to_double(ret);
      dashes[(num_dashes + 1)] = −1.0;      /* terminus */
      num_dashes += 2;
      p = (mp_dash_node) mp_link (p);
    }
```

d→*array* = *dashes*;
mp_dash_offset(*mp*, &*dashoff*, *h*);
take_scaled(*ret*, *dashoff*, *scf*);
d→*offset* = *number_to_double*(*ret*);
free_number(*ret*);
free_number(*arg1*);
}
free_number(*dashoff*);
free_number(*scf*);
return d;
}

490.   ⟨Copy the bounding box information from h to hh and make *bblast*(*hh*) point into the new object
        list 490⟩ ≡
number_clone(*hh*→*minx*, *h*→*minx*);
number_clone(*hh*→*miny*, *h*→*miny*);
number_clone(*hh*→*maxx*, *h*→*maxx*);
number_clone(*hh*→*maxy*, *h*→*maxy*);
hh→*bbtype* = *h*→*bbtype*;
p = (**mp_dash_node**) *edge_list*(*h*);
pp = (**mp_dash_node**) *edge_list*(*hh*);
while ((p ≠ (**mp_dash_node**) *bblast*(*h*))) {
   if (p ≡ Λ) *mp_confusion*(*mp*, "`bblast`");
   ;
   p = (**mp_dash_node**) *mp_link*(*p*);
   pp = (**mp_dash_node**) *mp_link*(*pp*);
}
bblast(*hh*) = (**mp_node**) *pp*

This code is used in section 487.

491.   Here is the promised routine for copying graphical objects into a new edge structure. It starts
copying at object p and stops just before object q. If q is NULL, it copies the entire sublist headed at p.
The resulting edge structure requires further initialization by *init_bbox*.

⟨Declarations 8⟩ +≡
   static mp_edge_header_node *mp_copy_objects*(**MP** *mp*, **mp_node** *p*, **mp_node** *q*);

492.   **mp_edge_header_node** *mp_copy_objects*(**MP** *mp*, **mp_node** *p*, **mp_node** *q*)
  {
    mp_edge_header_node *hh*;   /∗ the new edge header ∗/
    mp_node *pp*;   /∗ the last newly copied object ∗/
    quarterword *k* = 0;   /∗ temporary register ∗/
    hh = *mp_get_edge_header_node*(*mp*);
    set_dash_list(*hh*, *mp→null_dash*);
    edge_ref_count(*hh*) = 0;
    pp = *edge_list*(*hh*);
    while (*p* ≠ *q*) {
      ⟨Make *mp_link*(*pp*) point to a copy of object *p*, and update *p* and *pp* 493⟩;
    }
    obj_tail(*hh*) = *pp*;
    mp_link(*pp*) = Λ;
    return *hh*;
  }

493.   ⟨Make *mp_link*(*pp*) point to a copy of object *p*, and update *p* and *pp* 493⟩ ≡
  {
    switch (*mp_type*(*p*)) {
    case *mp_start_clip_node_type*: *k* = *start_clip_size*;
      break;
    case *mp_start_bounds_node_type*: *k* = *start_bounds_size*;
      break;
    case *mp_fill_node_type*: *k* = *fill_node_size*;
      break;
    case *mp_stroked_node_type*: *k* = *stroked_node_size*;
      break;
    case *mp_text_node_type*: *k* = *text_node_size*;
      break;
    case *mp_stop_clip_node_type*: *k* = *stop_clip_size*;
      break;
    case *mp_stop_bounds_node_type*: *k* = *stop_bounds_size*;
      break;
    default: /∗ there are no other valid cases, but please the compiler ∗/
      break;
    }
    mp_link(*pp*) = *malloc_node*((**size_t**) *k*);   /∗ *gr_object* ∗/
    pp = *mp_link*(*pp*);
    memcpy(*pp*, *p*, (**size_t**) *k*);
    pp→link = Λ;
    ⟨Fix anything in graphical object *pp* that should differ from the corresponding field in *p* 494⟩;
    p = *mp_link*(*p*);
  }
This code is used in section 492.

494. ⟨Fix anything in graphical object *pp* that should differ from the corresponding field in *p* 494⟩ ≡

 switch (*mp_type*(*p*)) {

 case *mp_start_clip_node_type*:

  {

   mp_start_clip_node *tt* = (**mp_start_clip_node**) *pp*;

   mp_start_clip_node *t* = (**mp_start_clip_node**) *p*;

   mp_path_p(*tt*) = *mp_copy_path*(*mp*, *mp_path_p*(*t*));

  }

  break;

 case *mp_start_bounds_node_type*:

  {

   mp_start_bounds_node *tt* = (**mp_start_bounds_node**) *pp*;

   mp_start_bounds_node *t* = (**mp_start_bounds_node**) *p*;

   mp_path_p(*tt*) = *mp_copy_path*(*mp*, *mp_path_p*(*t*));

  }

  break;

 case *mp_fill_node_type*:

  {

   mp_fill_node *tt* = (**mp_fill_node**) *pp*;

   mp_fill_node *t* = (**mp_fill_node**) *p*;

   new_number(*tt*→*red*);

   number_clone(*tt*→*red*, *t*→*red*);

   new_number(*tt*→*green*);

   number_clone(*tt*→*green*, *t*→*green*);

   new_number(*tt*→*blue*);

   number_clone(*tt*→*blue*, *t*→*blue*);

   new_number(*tt*→*black*);

   number_clone(*tt*→*black*, *t*→*black*);

   new_number(*tt*→*miterlim*);

   number_clone(*tt*→*miterlim*, *t*→*miterlim*);

   mp_path_p(*tt*) = *mp_copy_path*(*mp*, *mp_path_p*(*t*));

   if (*mp_pre_script*(*p*) ≠ Λ) *add_str_ref*(*mp_pre_script*(*p*));

   if (*mp_post_script*(*p*) ≠ Λ) *add_str_ref*(*mp_post_script*(*p*));

   if (*mp_pen_p*(*t*) ≠ Λ) *mp_pen_p*(*tt*) = *copy_pen*(*mp_pen_p*(*t*));

  }

  break;

 case *mp_stroked_node_type*:

  {

   mp_stroked_node *tt* = (**mp_stroked_node**) *pp*;

   mp_stroked_node *t* = (**mp_stroked_node**) *p*;

   new_number(*tt*→*red*);

   number_clone(*tt*→*red*, *t*→*red*);

   new_number(*tt*→*green*);

   number_clone(*tt*→*green*, *t*→*green*);

   new_number(*tt*→*blue*);

   number_clone(*tt*→*blue*, *t*→*blue*);

   new_number(*tt*→*black*);

   number_clone(*tt*→*black*, *t*→*black*);

   new_number(*tt*→*miterlim*);

   number_clone(*tt*→*miterlim*, *t*→*miterlim*);

   new_number(*tt*→*dash_scale*);

```
      number_clone(tt→dash_scale, t→dash_scale);
      if (mp_pre_script(p) ≠ Λ) add_str_ref(mp_pre_script(p));
      if (mp_post_script(p) ≠ Λ) add_str_ref(mp_post_script(p));
      mp_path_p(tt) = mp_copy_path(mp, mp_path_p(t));
      mp_pen_p(tt) = copy_pen(mp_pen_p(t));
      if (mp_dash_p(p) ≠ Λ) add_edge_ref(mp_dash_p(pp));
    }
    break;
  case mp_text_node_type:
    {
      mp_text_node tt = (mp_text_node) pp;
      mp_text_node t = (mp_text_node) p;

      new_number(tt→red);
      number_clone(tt→red, t→red);
      new_number(tt→green);
      number_clone(tt→green, t→green);
      new_number(tt→blue);
      number_clone(tt→blue, t→blue);
      new_number(tt→black);
      number_clone(tt→black, t→black);
      new_number(tt→width);
      number_clone(tt→width, t→width);
      new_number(tt→height);
      number_clone(tt→height, t→height);
      new_number(tt→depth);
      number_clone(tt→depth, t→depth);
      new_number(tt→tx);
      number_clone(tt→tx, t→tx);
      new_number(tt→ty);
      number_clone(tt→ty, t→ty);
      new_number(tt→txx);
      number_clone(tt→txx, t→txx);
      new_number(tt→tyx);
      number_clone(tt→tyx, t→tyx);
      new_number(tt→txy);
      number_clone(tt→txy, t→txy);
      new_number(tt→tyy);
      number_clone(tt→tyy, t→tyy);
      if (mp_pre_script(p) ≠ Λ) add_str_ref(mp_pre_script(p));
      if (mp_post_script(p) ≠ Λ) add_str_ref(mp_post_script(p));
      add_str_ref(mp_text_p(pp));
    }
    break;
  case mp_stop_clip_node_type: case mp_stop_bounds_node_type: break;
  default:      /* there are no other valid cases, but please the compiler */
    break;
  }
```

This code is used in section 493.

495.   Here is one way to find an acceptable value for the second argument to *copy_objects*. Given a non-NULL graphical object list, *skip_1component* skips past one picture component, where a "picture component" is a single graphical object, or a start bounds or start clip object and everything up through the matching stop bounds or stop clip object.

```
static mp_node mp_skip_1component(MP mp, mp_node p)
{
    integer lev;      /* current nesting level */
    lev = 0;
    (void) mp;
    do {
        if (is_start_or_stop(p)) {
            if (is_stop(p))  decr(lev);
            else  incr(lev);
        }
        p = mp_link(p);
    } while (lev ≠ 0);
    return p;
}
```

496.   Here is a diagnostic routine for printing an edge structure in symbolic form.

⟨ Declarations 8 ⟩ +≡
```
    static void mp_print_edges(MP mp, mp_node h, const char *s, boolean nuline);
```

497.   **void** *mp_print_edges*(**MP** *mp*, **mp_node** *h*, **const char** *∗s*, **boolean** *nuline*)
```
{
    mp_node p;      /* a graphical object to be printed */
    mp_number scf;      /* a scale factor for the dash pattern */
    boolean ok_to_dash;      /* false for polygonal pen strokes */
    new_number(scf);
    mp_print_diagnostic(mp, "Edge␣structure", s, nuline);
    p = edge_list(h);
    while (mp_link(p) ≠ Λ) {
        p = mp_link(p);
        mp_print_ln(mp);
        switch (mp_type(p)) {
            ⟨ Cases for printing graphical object node p 498 ⟩;
            default: mp_print(mp, "[unknown␣object␣type!]");
                break;
        }
    }
    mp_print_nl(mp, "End␣edges");
    if (p ≠ obj_tail(h))  mp_print(mp, "?");
    ;
    mp_end_diagnostic(mp, true);
    free_number(scf);
}
```

498.   ⟨Cases for printing graphical object node p 498⟩ ≡
case $mp\_fill\_node\_type$: $mp\_print(mp,$ "Filled␣contour␣");
  $mp\_print\_obj\_color(mp, p);$
  $mp\_print\_char(mp, xord(\text{':'}));$
  $mp\_print\_ln(mp);$
  $mp\_pr\_path(mp, mp\_path\_p((\textbf{mp\_fill\_node}) \, p));$
  $mp\_print\_ln(mp);$
  if $((mp\_pen\_p((\textbf{mp\_fill\_node}) \, p) \neq \Lambda))$ {
    ⟨Print join type for graphical object p 499⟩;
    $mp\_print(mp,$ "␣with␣pen");
    $mp\_print\_ln(mp);$
    $mp\_pr\_pen(mp, mp\_pen\_p((\textbf{mp\_fill\_node}) \, p));$
  }
  break;
See also sections 503, 507, 508, and 509.

This code is used in section 497.

499.   ⟨Print join type for graphical object p 499⟩ ≡
  switch $(((\textbf{mp\_stroked\_node}) \, p)\text{-}ljoin)$ {
  case 0: $mp\_print(mp,$ "mitered␣joins␣limited␣");
    $print\_number(((\textbf{mp\_stroked\_node}) \, p)\text{-}miterlim);$
    break;
  case 1: $mp\_print(mp,$ "round␣joins");
    break;
  case 2: $mp\_print(mp,$ "beveled␣joins");
    break;
  default: $mp\_print(mp,$ "??␣joins");
    ;
    break;
  }
This code is used in sections 498 and 500.

500.   For stroked nodes, we need to print $lcap\_val(p)$ as well.

⟨Print join and cap types for stroked node p 500⟩ ≡
  switch $(((\textbf{mp\_stroked\_node}) \, p)\text{-}lcap)$ {
  case 0: $mp\_print(mp,$ "butt");
    break;
  case 1: $mp\_print(mp,$ "round");
    break;
  case 2: $mp\_print(mp,$ "square");
    break;
  default: $mp\_print(mp,$ "??");
    break;
  }
  $mp\_print(mp,$ "␣ends,␣"); ⟨Print join type for graphical object p 499⟩
This code is used in section 503.

501.   Here is a routine that prints the color of a graphical object if it isn't black (the default color).

⟨Declarations 8⟩ +≡
  static void $mp\_print\_obj\_color(\textbf{MP} \; mp, \textbf{mp\_node} \; p);$

502.   **void** $mp\_print\_obj\_color(\mathbf{MP}\ mp, \mathbf{mp\_node}\ p)$
{
  $\mathbf{mp\_stroked\_node}\ p0 = (\mathbf{mp\_stroked\_node})\ p;$
  if $(\mathbf{mp\_color\_model}(p) \equiv mp\_grey\_model)$ {
    if $(number\_positive(p0\rightarrow grey))$ {
      $mp\_print(mp, \texttt{"greyed}\sqcup\texttt{"});$
      $mp\_print\_char(mp, xord(\texttt{'('}));$
      $print\_number(p0\rightarrow grey);$
      $mp\_print\_char(mp, xord(\texttt{')'}));$
    }
    ;
  }
  else if $(\mathbf{mp\_color\_model}(p) \equiv mp\_cmyk\_model)$ {
    if $(number\_positive(p0\rightarrow cyan) \vee number\_positive(p0\rightarrow magenta) \vee number\_positive(p0\rightarrow yellow) \vee$
        $number\_positive(p0\rightarrow black))$ {
      $mp\_print(mp, \texttt{"processcolored}\sqcup\texttt{"});$
      $mp\_print\_char(mp, xord(\texttt{'('}));$
      $print\_number(p0\rightarrow cyan);$
      $mp\_print\_char(mp, xord(\texttt{','}));$
      $print\_number(p0\rightarrow magenta);$
      $mp\_print\_char(mp, xord(\texttt{','}));$
      $print\_number(p0\rightarrow yellow);$
      $mp\_print\_char(mp, xord(\texttt{','}));$
      $print\_number(p0\rightarrow black);$
      $mp\_print\_char(mp, xord(\texttt{')'}));$
    }
    ;
  }
  else if $(\mathbf{mp\_color\_model}(p) \equiv mp\_rgb\_model)$ {
    if $(number\_positive(p0\rightarrow red) \vee number\_positive(p0\rightarrow green) \vee number\_positive(p0\rightarrow blue))$ {
      $mp\_print(mp, \texttt{"colored}\sqcup\texttt{"});$
      $mp\_print\_char(mp, xord(\texttt{'('}));$
      $print\_number(p0\rightarrow red);$
      $mp\_print\_char(mp, xord(\texttt{','}));$
      $print\_number(p0\rightarrow green);$
      $mp\_print\_char(mp, xord(\texttt{','}));$
      $print\_number(p0\rightarrow blue);$
      $mp\_print\_char(mp, xord(\texttt{')'}));$
    }
    ;
  }
}

503.   ⟨Cases for printing graphical object node p 498⟩ +≡
case $mp\_stroked\_node\_type$: $mp\_print(mp, \texttt{"Filled}_\sqcup\texttt{pen}_\sqcup\texttt{stroke}_\sqcup\texttt{"})$;
  $mp\_print\_obj\_color(mp, p)$;
  $mp\_print\_char(mp, xord(\texttt{':'}))$;
  $mp\_print\_ln(mp)$;
  $mp\_pr\_path(mp, mp\_path\_p((\textbf{mp\_stroked\_node})\ p))$;
  if $(mp\_dash\_p(p) \neq \Lambda)$ {
    $mp\_print\_nl(mp, \texttt{"dashed}_\sqcup\texttt{("})$;
    ⟨Finish printing the dash pattern that p refers to 504⟩;
  }
  $mp\_print\_ln(mp)$;
  ⟨Print join and cap types for stroked node p 500⟩;
  $mp\_print(mp, \texttt{"}_\sqcup\texttt{with}_\sqcup\texttt{pen"})$;
  $mp\_print\_ln(mp)$;
  if $(mp\_pen\_p((\textbf{mp\_stroked\_node})\ p) \equiv \Lambda)$ {
    $mp\_print(mp, \texttt{"???"})$;   /∗ shouldn't happen ∗/
  }
  else {
    $mp\_pr\_pen(mp, mp\_pen\_p((\textbf{mp\_stroked\_node})\ p))$;
  }
  break;

504.   Normally, the *dash_list* field in an edge header is set to *null_dash* when it is not known to define a suitable dash pattern. This is disallowed here because the *mp_dash_p* field should never point to such an edge header. Note that memory is allocated for *start_x*(*null_dash*) and we are free to give it any convenient value.

⟨Finish printing the dash pattern that *p* refers to 504⟩ ≡

```
{
    mp_dash_node ppd, hhd;

    ok_to_dash = pen_is_elliptical(mp_pen_p((mp_stroked_node) p));
    if (¬ok_to_dash) set_number_to_unity(scf);
    else number_clone(scf, ((mp_stroked_node) p)→dash_scale);
    hhd = (mp_dash_node) mp_dash_p(p);
    ppd = dash_list(hhd);
    if ((ppd ≡ mp→null_dash) ∨ number_negative(hhd→dash_y)) {
        mp_print(mp, "␣??");
    }
    else {
        mp_number dashoff;
        mp_number ret, arg1;

        new_number(ret);
        new_number(arg1);
        new_number(dashoff);
        set_number_from_addition(mp→null_dash→start_x, ppd→start_x, hhd→dash_y);
        while (ppd ≠ mp→null_dash) {
            mp_print(mp, "on␣");
            set_number_from_substraction(arg1, ppd→stop_x, ppd→start_x);
            take_scaled(ret, arg1, scf);
            print_number(ret);
            mp_print(mp, "␣off␣");
            set_number_from_substraction(arg1, ((mp_dash_node) mp_link(ppd))→start_x, ppd→stop_x);
            take_scaled(ret, arg1, scf);
            print_number(ret);
            ppd = (mp_dash_node) mp_link(ppd);
            if (ppd ≠ mp→null_dash) mp_print_char(mp, xord('␣'));
        }
        mp_print(mp, ")␣shifted␣");
        mp_dash_offset(mp, &dashoff, hhd);
        take_scaled(ret, dashoff, scf);
        number_negate(ret);
        print_number(ret);
        free_number(dashoff);
        free_number(ret);
        free_number(arg1);
        if (¬ok_to_dash ∨ number_zero(hhd→dash_y)) mp_print(mp, "␣(this␣will␣be␣ignored)");
    }
}
```

This code is used in section 503.

505.   ⟨Declarations 8⟩ +≡

```
static void mp_dash_offset(MP mp, mp_number *x, mp_dash_node h);
```

506.   **void** $mp\_dash\_offset(\textbf{MP}\ mp, \textbf{mp\_number}\ *x, \textbf{mp\_dash\_node}\ h)$
{
    if $(dash\_list(h) \equiv mp\text{-}null\_dash \lor number\_negative(h\text{-}dash\_y))$ $mp\_confusion(mp, \texttt{"dash0"})$;
    ;
    if $(number\_zero(h\text{-}dash\_y))$ {
        $set\_number\_to\_zero(*x)$;
    }
    else {
        $number\_clone(*x, (dash\_list(h))\text{-}start\_x)$;
        $number\_modulo(*x, h\text{-}dash\_y)$;
        $number\_negate(*x)$;
        if $(number\_negative(*x))$ $number\_add(*x, h\text{-}dash\_y)$;
    }
}

507.   ⟨ Cases for printing graphical object node p 498 ⟩ +≡
case $mp\_text\_node\_type$:
    {
        $\textbf{mp\_text\_node}\ p0 = (\textbf{mp\_text\_node})\ p$;

        $mp\_print\_char(mp, xord(\texttt{'"'}))$;
        $mp\_print\_str(mp, mp\_text\_p(p))$;
        $mp\_print(mp, \texttt{"\textbackslash"}_\sqcup\texttt{infont}_\sqcup\texttt{\textbackslash""})$;
        $mp\_print(mp, mp\text{-}font\_name[mp\_font\_n(p)])$;
        $mp\_print\_char(mp, xord(\texttt{'"'}))$;
        $mp\_print\_ln(mp)$;
        $mp\_print\_obj\_color(mp, p)$;
        $mp\_print(mp, \texttt{"transformed}_\sqcup\texttt{"})$;
        $mp\_print\_char(mp, xord(\texttt{'('}))$;
        $print\_number(p0\text{-}tx)$;
        $mp\_print\_char(mp, xord(\texttt{','}))$;
        $print\_number(p0\text{-}ty)$;
        $mp\_print\_char(mp, xord(\texttt{','}))$;
        $print\_number(p0\text{-}txx)$;
        $mp\_print\_char(mp, xord(\texttt{','}))$;
        $print\_number(p0\text{-}txy)$;
        $mp\_print\_char(mp, xord(\texttt{','}))$;
        $print\_number(p0\text{-}tyx)$;
        $mp\_print\_char(mp, xord(\texttt{','}))$;
        $print\_number(p0\text{-}tyy)$;
        $mp\_print\_char(mp, xord(\texttt{')'}))$;
    }
    break;

508.   ⟨ Cases for printing graphical object node p 498 ⟩ +≡
case $mp\_start\_clip\_node\_type$: $mp\_print(mp, \texttt{"clipping}_\sqcup\texttt{path:"})$;
    $mp\_print\_ln(mp)$;
    $mp\_pr\_path(mp, mp\_path\_p((\textbf{mp\_start\_clip\_node})\ p))$;
    break;
case $mp\_stop\_clip\_node\_type$: $mp\_print(mp, \texttt{"stop}_\sqcup\texttt{clipping"})$;
    break;

509.   ⟨Cases for printing graphical object node p 498⟩ +≡
case $mp\_start\_bounds\_node\_type$: $mp\_print(mp,$ `"setbounds⎵path:"`$)$;
  $mp\_print\_ln(mp)$;
  $mp\_pr\_path(mp, mp\_path\_p((\textbf{mp\_start\_bounds\_node})\ p))$;
  break;
case $mp\_stop\_bounds\_node\_type$: $mp\_print(mp,$ `"end⎵of⎵setbounds"`$)$;
  break;

510.   To initialize the *dash_list* field in an edge header h, we need a subroutine that scans an edge structure and tries to interpret it as a dash pattern. This can only be done when there are no filled regions or clipping paths and all the pen strokes have the same color. The first step is to let y_0 be the initial y coordinate of the first pen stroke. Then we implicitly project all the pen stroke paths onto the line $y = y_0$ and require that there be no retracing. If the resulting paths cover a range of x coordinates of length Δx, we set $dash\_y(h)$ to the length of the dash pattern by finding the maximum of Δx and the absolute value of y_0.

  static mp_edge_header_node $mp\_make\_dashes(\textbf{MP}\ mp, \textbf{mp\_edge\_header\_node}\ h)$
  $\{$ /* returns h or Λ */
    mp_node p;   /* this scans the stroked nodes in the object list */
    mp_node $p0$;   /* if not Λ this points to the first stroked node */
    mp_knot pp, qq, rr;   /* pointers into $mp\_path\_p(p)$ */
    mp_dash_node d, dd;   /* pointers used to create the dash list */
    mp_number $y0$;

    ⟨Other local variables in *make_dashes* 521⟩;
    if $(dash\_list(h) \neq mp{\rightarrow}null\_dash)$ **return** h;
    $new\_number(y0)$;   /* the initial y coordinate */
    $p0 = \Lambda$;
    $p = mp\_link(edge\_list(h))$;
    while $(p \neq \Lambda)$ $\{$
      if $(mp\_type(p) \neq mp\_stroked\_node\_type)$ $\{$
        ⟨Compain that the edge structure contains a node of the wrong type and **goto** *not_found* 511⟩;
      $\}$
      $pp = mp\_path\_p((\textbf{mp\_stroked\_node})\ p)$;
      if $(p0 \equiv \Lambda)$ $\{$
        $p0 = p$;
        $number\_clone(y0, pp{\rightarrow}y\_coord)$;
      $\}$
      ⟨Make d point to a new dash node created from stroke p and path pp or **goto** *not_found* if there is an error 514⟩;
      ⟨Insert d into the dash list and **goto** *not_found* if there is an error 517⟩;
      $p = mp\_link(p)$;
    $\}$
    if $(dash\_list(h) \equiv mp{\rightarrow}null\_dash)$ **goto** `NOT_FOUND`;   /* No error message */
    ⟨Scan $dash\_list(h)$ and deal with any dashes that are themselves dashed 520⟩;
    ⟨Set $dash\_y(h)$ and merge the first and last dashes if necessary 518⟩;
    $free\_number(y0)$;
    return h;
  `NOT_FOUND`: $free\_number(y0)$;
    ⟨Flush the dash list, recycle h and return Λ 519⟩;
  $\}$

511. ⟨Compain that the edge structure contains a node of the wrong type and **goto** *not_found* 511⟩ ≡
  {
    const char *∗hlp*[ ] = {"When␣you␣say␣'dashed␣p',␣picture␣p␣should␣not␣contain␣any",
        "text,␣filled␣regions,␣or␣clipping␣paths.␣␣This␣time␣it␣did",
        "so␣I'll␣just␣make␣it␣a␣solid␣line␣instead.", Λ};

    mp_back_error(*mp*, "Picture␣is␣too␣complicated␣to␣use␣as␣a␣dash␣pattern", *hlp*, *true*);
    mp_get_x_next(*mp*);
    goto NOT_FOUND;
  }

This code is used in section 510.

512. A similar error occurs when monotonicity fails.

⟨Declarations 8⟩ +≡
  static void *mp_x_retrace_error*(**MP** *mp*);

513. **void** *mp_x_retrace_error*(**MP** *mp*)
  {
    const char *∗hlp*[ ] = {"When␣you␣say␣'dashed␣p',␣every␣path␣in␣p␣should␣be␣monotone",
        "in␣x␣and␣there␣must␣be␣no␣overlapping.␣␣This␣failed",
        "so␣I'll␣just␣make␣it␣a␣solid␣line␣instead.", Λ};

    mp_back_error(*mp*, "Picture␣is␣too␣complicated␣to␣use␣as␣a␣dash␣pattern", *hlp*, *true*);
    mp_get_x_next(*mp*);
  }

514. We stash *p* in *dash_info*(*d*) if *mp_dash_p*(*p*) <> 0 so that subsequent processing can handle the case where the pen stroke *p* is itself dashed.

#define *dash_info*(*A*) ((**mp_dash_node**)(*A*))⃗*dash_info_*
        /∗ in an edge header this points to the first dash node ∗/

⟨Make *d* point to a new dash node created from stroke *p* and path *pp* or **goto** *not_found* if there is an
        error 514⟩ ≡
  ⟨Make sure *p* and *p0* are the same color and **goto** *not_found* if there is an error 516⟩;
  rr = *pp*;
  if (*mp_next_knot*(*pp*) ≠ *pp*) {
    do {
      qq = *rr*;
      rr = *mp_next_knot*(*rr*);
      ⟨Check for retracing between knots *qq* and *rr* and **goto** *not_found* if there is a problem 515⟩;
    } **while** (*mp_right_type*(*rr*) ≠ *mp_endpoint*);
  }
  d = (**mp_dash_node**) *mp_get_dash_node*(*mp*);
  if (*mp_dash_p*(*p*) ≡ Λ) *dash_info*(*d*) = Λ;
  else *dash_info*(*d*) = *p*;
  if (*number_less*(*pp*⃗*x_coord*, *rr*⃗*x_coord*)) {
    number_clone(*d*⃗*start_x*, *pp*⃗*x_coord*);
    number_clone(*d*⃗*stop_x*, *rr*⃗*x_coord*);
  }
  else {
    number_clone(*d*⃗*start_x*, *rr*⃗*x_coord*);
    number_clone(*d*⃗*stop_x*, *pp*⃗*x_coord*);
  }

This code is used in section 510.

515.   We also need to check for the case where the segment from qq to rr is monotone in x but is reversed relative to the path from pp to qq.

$\langle$ Check for retracing between knots qq and rr and **goto** *not_found* if there is a problem 515 $\rangle \equiv$
```
{
   mp_number x0, x1, x2, x3;     /* x coordinates of the segment from qq to rr */
   new_number(x0);
   new_number(x1);
   new_number(x2);
   new_number(x3);
   number_clone(x0, qq→x_coord);
   number_clone(x1, qq→right_x);
   number_clone(x2, rr→left_x);
   number_clone(x3, rr→x_coord);
   if (number_greater(x0, x1) ∨ number_greater(x1, x2) ∨ number_greater(x2, x3)) {
      if (number_less(x0, x1) ∨ number_less(x1, x2) ∨ number_less(x2, x3)) {
         mp_number a1, a2, a3, a4;
         mp_number test;

         new_number(test);
         new_number(a1);
         new_number(a2);
         new_number(a3);
         new_number(a4);
         set_number_from_substraction(a1, x2, x1);
         set_number_from_substraction(a2, x2, x1);
         set_number_from_substraction(a3, x1, x0);
         set_number_from_substraction(a4, x3, x2);
         ab_vs_cd(test, a1, a2, a3, a4);
         free_number(a1);
         free_number(a2);
         free_number(a3);
         free_number(a4);
         if (number_positive(test)) {
            mp_x_retrace_error(mp);
            free_number(x0);
            free_number(x1);
            free_number(x2);
            free_number(x3);
            free_number(test);
            goto NOT_FOUND;
         }
         free_number(test);
      }
   }
   if (number_greater(pp→x_coord, x0) ∨ number_greater(x0, x3)) {
      if (number_less(pp→x_coord, x0) ∨ number_less(x0, x3)) {
         mp_x_retrace_error(mp);
         free_number(x0);
         free_number(x1);
         free_number(x2);
         free_number(x3);
         goto NOT_FOUND;
```

```
        }
    }
    free_number(x0);
    free_number(x1);
    free_number(x2);
    free_number(x3);
}
```

This code is used in section 514.

516.   ⟨ Make sure p and $p0$ are the same color and **goto** *not_found* if there is an error 516 ⟩ ≡
  if ($¬number\_equal$(((**mp_stroked_node**) p)$→red$,
        ((**mp_stroked_node**) $p0$)$→red$) ∨ $¬number\_equal$(((**mp_stroked_node**) p)$→black$,
        ((**mp_stroked_node**) $p0$)$→black$) ∨ $¬number\_equal$(((**mp_stroked_node**)
        p)$→green$, ((**mp_stroked_node**) $p0$)$→green$) ∨ $¬number\_equal$(((**mp_stroked_node**)
        p)$→blue$, ((**mp_stroked_node**) $p0$)$→blue$)) {
    const char ∗hlp[] = {"When␣you␣say␣'dashed␣p',␣everything␣in␣picture␣p␣should",
        "be␣the␣same␣color.␣␣␣I␣can\'t␣handle␣your␣color␣changes",
        "so␣I'll␣just␣make␣it␣a␣solid␣line␣instead.", Λ};

    mp_back_error(mp, "Picture␣is␣too␣complicated␣to␣use␣as␣a␣dash␣pattern", hlp, $true$);
    $mp\_get\_x\_next$(mp);
    goto NOT_FOUND;
  }

This code is used in section 514.

517.   ⟨ Insert d into the dash list and **goto** *not_found* if there is an error 517 ⟩ ≡
  $number\_clone$($mp→null\_dash→start\_x$, $d→stop\_x$);
  dd = (**mp_dash_node**) h;   /∗ this makes $mp\_link(dd) = dash\_list(h)$ ∗/
  while ($number\_less$(((**mp_dash_node**) $mp\_link(dd)$)$→start\_x$, $d→stop\_x$))
    dd = (**mp_dash_node**) $mp\_link(dd)$;
  if ($dd ≠$ (**mp_dash_node**) h) {
    if ($number\_greater$($dd→stop\_x$, $d→start\_x$)) {
      $mp\_x\_retrace\_error$(mp);
      goto NOT_FOUND;
    }
    ;
  }
  $mp\_link(d) = mp\_link(dd)$; $mp\_link(dd)$ = (**mp_node**) d

This code is used in section 510.

518.   ⟨Set *dash_y*(*h*) and merge the first and last dashes if necessary 518⟩ ≡

 d = *dash_list*(*h*);

 while ((*mp_link*(*d*) ≠ (**mp_node**) *mp→null_dash*)) *d* = (**mp_dash_node**) *mp_link*(*d*);

 dd = *dash_list*(*h*);

 set_number_from_substraction(*h→dash_y*, *d→stop_x*, *dd→start_x*);

 {

  mp_number *absval*;

  new_number(*absval*);

  number_clone(*absval*, *y0*);

  number_abs(*absval*);

  if (*number_greater*(*absval*, *h→dash_y*)) {

   number_clone(*h→dash_y*, *absval*);

  }

  else if (*d* ≠ *dd*) {

   set_dash_list(*h*, *mp_link*(*dd*));

   set_number_from_addition(*d→stop_x*, *dd→stop_x*, *h→dash_y*);

   mp_free_node(*mp*, (**mp_node**) *dd*, *dash_node_size*);

  }

  free_number(*absval*);

 }

This code is used in section 510.

519.   We get here when the argument is a NULL picture or when there is an error. Recovering from an error involves making *dash_list*(*h*) empty to indicate that *h* is not known to be a valid dash pattern. We also dereference *h* since it is not being used for the return value.

⟨Flush the dash list, recycle *h* and return Λ 519⟩ ≡

 mp_flush_dash_list(*mp*, *h*);

 delete_edge_ref(*h*); **return** Λ

This code is used in section 510.

520.   Having carefully saved the dashed stroked nodes in the corresponding dash nodes, we must be prepared to break up these dashes into smaller dashes.

⟨ Scan *dash_list*(*h*) and deal with any dashes that are themselves dashed 520 ⟩ ≡

```
{
    mp_number hsf;      /* the dash pattern from hh gets scaled by this */
    new_number(hsf);
    d = (mp_dash_node) h;      /* now mp_link(d) = dash_list(h) */
    while (mp_link(d) ≠ (mp_node) mp→null_dash) {
        ds = dash_info(mp_link(d));
        if (ds ≡ Λ) {
            d = (mp_dash_node) mp_link(d);
        }
        else {
            hh = (mp_edge_header_node) mp_dash_p(ds);
            number_clone(hsf, ((mp_stroked_node) ds)→dash_scale);
            if (hh ≡ Λ) mp_confusion(mp, "dash1");
            ;      /* clang: dereference null pointer 'hh' */
            assert(hh);
            if (number_zero(((mp_dash_node) hh)→dash_y)) {
                d = (mp_dash_node) mp_link(d);
            }
            else {
                if (dash_list(hh) ≡ Λ) mp_confusion(mp, "dash1");
                ;
                ⟨Replace mp_link(d) by a dashed version as determined by edge header hh and scale factor
                    ds 522⟩;
            }
        }
    }
    free_number(hsf);
}
```

This code is used in section 510.

521.   ⟨ Other local variables in *make_dashes* 521 ⟩ ≡

```
    mp_dash_node dln;      /* mp_link(d) */
    mp_edge_header_node hh;      /* an edge header that tells how to break up dln */
    mp_node ds;      /* the stroked node from which hh and hsf are derived */
```

This code is used in section 510.

522.   ⟨Replace $mp\_link(d)$ by a dashed version as determined by edge header hh and scale factor ds 522⟩ ≡

```
{
    mp_number xoff;        /∗ added to x values in dash_list(hh) to match dln ∗/
    mp_number dashoff;
    mp_number r1, r2;
    new_number(r1);
    new_number(r2);
    dln = (mp_dash_node) mp_link(d);
    dd = dash_list(hh);        /∗ clang: dereference null pointer 'dd' ∗/
    assert(dd);
    new_number(xoff);
    new_number(dashoff);
    mp_dash_offset(mp, &dashoff, (mp_dash_node) hh);
    take_scaled(r1, hsf, dd→start_x);
    take_scaled(r2, hsf, dashoff);
    number_add(r1, r2);
    set_number_from_substraction(xoff, dln→start_x, r1);
    free_number(dashoff);
    take_scaled(r1, hsf, dd→start_x);
    take_scaled(r2, hsf, hh→dash_y);
    set_number_from_addition(mp→null_dash→start_x, r1, r2);
    number_clone(mp→null_dash→stop_x, mp→null_dash→start_x);
    ⟨Advance dd until finding the first dash that overlaps dln when offset by xoff 523⟩;
    while (number_lessequal(dln→start_x, dln→stop_x)) {
        ⟨If dd has 'fallen off the end', back up to the beginning and fix xoff 524⟩;
        ⟨Insert a dash between d and dln for the overlap with the offset version of dd 525⟩;
        dd = (mp_dash_node) mp_link(dd);
        take_scaled(r1, hsf, dd→start_x);
        set_number_from_addition(dln→start_x, xoff, r1);
    }
    free_number(xoff);
    free_number(r1);
    free_number(r2);
    mp_link(d) = mp_link(dln);
    mp_free_node(mp, (mp_node) dln, dash_node_size);
}
```

This code is used in section 520.

523.   The name of this module is a bit of a lie because we just find the first *dd* where *take_scaled*(*hsf*, *stop_x*(*dd*))▮ is large enough to make an overlap possible. It could be that the unoffset version of dash *dln* falls in the gap between *dd* and its predecessor.

⟨ Advance *dd* until finding the first dash that overlaps *dln* when offset by *xoff* 523 ⟩ ≡
```
  {
    mp_number r1;

    new_number(r1);
    take_scaled(r1, hsf, dd→stop_x);
    number_add(r1, xoff);
    while (number_less(r1, dln→start_x)) {
      dd = (mp_dash_node) mp_link(dd);
      take_scaled(r1, hsf, dd→stop_x);
      number_add(r1, xoff);
    }
    free_number(r1);
  }
```
This code is used in section 522.

524.   ⟨ If *dd* has 'fallen off the end', back up to the beginning and fix *xoff* 524 ⟩ ≡
```
  if (dd ≡ mp→null_dash) {
    mp_number ret;

    new_number(ret);
    dd = dash_list(hh);
    take_scaled(ret, hsf, hh→dash_y);
    number_add(xoff, ret);
    free_number(ret);
  }
```
This code is used in section 522.

525.   At this point we already know that $start\_x(dln) \leq xoff + take\_scaled(hsf, stop\_x(dd))$.

⟨ Insert a dash between d and dln for the overlap with the offset version of dd 525 ⟩ ≡

```
{
  mp_number r1;

  new_number(r1);
  take_scaled(r1, hsf, dd→start_x);
  number_add(r1, xoff);
  if (number_lessequal(r1, dln→stop_x)) {
    mp_link(d) = (mp_node) mp_get_dash_node(mp);
    d = (mp_dash_node) mp_link(d);
    mp_link(d) = (mp_node) dln;
    take_scaled(r1, hsf, dd→start_x);
    number_add(r1, xoff);
    if (number_greater(dln→start_x, r1)) number_clone(d→start_x, dln→start_x);
    else {
      number_clone(d→start_x, r1);
    }
    take_scaled(r1, hsf, dd→stop_x);
    number_add(r1, xoff);
    if (number_less(dln→stop_x, r1)) number_clone(d→stop_x, dln→stop_x);
    else {
      number_clone(d→stop_x, r1);
    }
  }
  free_number(r1);
}
```

This code is used in section 522.

526.   The next major task is to update the bounding box information in an edge header h. This is done via a procedure *adjust_bbox* that enlarges an edge header's bounding box to accommodate the box computed by *path_bbox* or *pen_bbox*. (This is stored in global variables *minx*, *miny*, *maxx*, and *maxy*.)

```
static void mp_adjust_bbox(MP mp, mp_edge_header_node h)
{
  if (number_less(mp_minx, h→minx)) number_clone(h→minx, mp_minx);
  if (number_less(mp_miny, h→miny)) number_clone(h→miny, mp_miny);
  if (number_greater(mp_maxx, h→maxx)) number_clone(h→maxx, mp_maxx);
  if (number_greater(mp_maxy, h→maxy)) number_clone(h→maxy, mp_maxy);
}
```

527.   Here is a special routine for updating the bounding box information in edge header h to account for the squared-off ends of a non-cyclic path p that is to be stroked with the pen pp.

static void $mp\_box\_ends(\textbf{MP} \ mp, \textbf{mp\_knot} \ p, \textbf{mp\_knot} \ pp, \textbf{mp\_edge\_header\_node} \ h)$
{
   mp_knot q;   /∗ a knot node adjacent to knot p ∗/
   $mp\_fraction \ dx, dy$;   /∗ a unit vector in the direction out of the path at p ∗/
   mp_number d;   /∗ a factor for adjusting the length of (dx, dy) ∗/
   mp_number z;   /∗ a coordinate being tested against the bounding box ∗/
   mp_number xx, yy;   /∗ the extreme pen vertex in the (dx, dy) direction ∗/
   integer i;   /∗ a loop counter ∗/
   $new\_fraction(dx)$;
   $new\_fraction(dy)$;
   $new\_number(xx)$;
   $new\_number(yy)$;
   $new\_number(z)$;
   $new\_number(d)$;
   if $(mp\_right\_type(p) \neq mp\_endpoint)$ {
     $q = mp\_next\_knot(p)$;
     while (1) {
       ⟨ Make (dx, dy) the final direction for the path segment from q to p; set d 528 ⟩;
       $pyth\_add(d, dx, dy)$;
       if $(number\_positive(d))$ {
         ⟨ Normalize the direction (dx, dy) and find the pen offset (xx, yy) 529 ⟩;
         for $(i = 1; \ i \leq 2; \ i\text{++})$ {
           ⟨ Use (dx, dy) to generate a vertex of the square end cap and update the bounding box to
               accommodate it 530 ⟩;
           $number\_negate(dx)$;
           $number\_negate(dy)$;
         }
       }
       if $(mp\_right\_type(p) \equiv mp\_endpoint)$ {
         goto DONE;
       }
       else {
         ⟨ Advance p to the end of the path and make q the previous knot 531 ⟩;
       }
     }
   }
DONE: $free\_number(dx)$;
   $free\_number(dy)$;
   $free\_number(xx)$;
   $free\_number(yy)$;
   $free\_number(z)$;
   $free\_number(d)$;
}

528.   ⟨ Make (dx, dy) the final direction for the path segment from q to p; set d 528 ⟩ ≡
  if $(q \equiv mp\_next\_knot(p))$ {
    $set\_number\_from\_substraction(dx, p{\rightarrow}x\_coord, p{\rightarrow}right\_x);$
    $set\_number\_from\_substraction(dy, p{\rightarrow}y\_coord, p{\rightarrow}right\_y);$
    if $(number\_zero(dx) \wedge number\_zero(dy))$ {
      $set\_number\_from\_substraction(dx, p{\rightarrow}x\_coord, q{\rightarrow}left\_x);$
      $set\_number\_from\_substraction(dy, p{\rightarrow}y\_coord, q{\rightarrow}left\_y);$
    }
  }
  else {
    $set\_number\_from\_substraction(dx, p{\rightarrow}x\_coord, p{\rightarrow}left\_x);$
    $set\_number\_from\_substraction(dy, p{\rightarrow}y\_coord, p{\rightarrow}left\_y);$
    if $(number\_zero(dx) \wedge number\_zero(dy))$ {
      $set\_number\_from\_substraction(dx, p{\rightarrow}x\_coord, q{\rightarrow}right\_x);$
      $set\_number\_from\_substraction(dy, p{\rightarrow}y\_coord, q{\rightarrow}right\_y);$
    }
  }
  $set\_number\_from\_substraction(dx, p{\rightarrow}x\_coord, q{\rightarrow}x\_coord);$
  $set\_number\_from\_substraction(dy, p{\rightarrow}y\_coord, q{\rightarrow}y\_coord);$
This code is used in section 527.

529.   ⟨ Normalize the direction (dx, dy) and find the pen offset (xx, yy) 529 ⟩ ≡
  {
    mp_number $arg1$, r;
    $new\_fraction(r);$
    $new\_number(arg1);$
    $make\_fraction(r, dx, d);$
    $number\_clone(dx, r);$
    $make\_fraction(r, dy, d);$
    $number\_clone(dy, r);$
    $free\_number(r);$
    $number\_clone(arg1, dy);$
    $number\_negate(arg1);$
    $mp\_find\_offset(mp, arg1, dx, pp);$
    $free\_number(arg1);$
    $number\_clone(xx, mp{\rightarrow}cur\_x);$
    $number\_clone(yy, mp{\rightarrow}cur\_y);$
  }
This code is used in section 527.

530. ⟨Use (dx, dy) to generate a vertex of the square end cap and update the bounding box to
accommodate it 530⟩ ≡

```
{
  mp_number r1, r2, arg1;
  new_number(arg1);
  new_fraction(r1);
  new_fraction(r2);
  mp_find_offset(mp, dx, dy, pp);
  set_number_from_substraction(arg1, xx, mp→cur_x);
  take_fraction(r1, arg1, dx);
  set_number_from_substraction(arg1, yy, mp→cur_y);
  take_fraction(r2, arg1, dy);
  set_number_from_addition(d, r1, r2);
  if ((number_negative(d) ∧ (i ≡ 1)) ∨ (number_positive(d) ∧ (i ≡ 2)))  mp_confusion(mp, "box_ends");
  ;
  take_fraction(r1, d, dx);
  set_number_from_addition(z, p→x_coord, mp→cur_x);
  number_add(z, r1);
  if (number_less(z, h→minx))  number_clone(h→minx, z);
  if (number_greater(z, h→maxx))  number_clone(h→maxx, z);
  take_fraction(r1, d, dy);
  set_number_from_addition(z, p→y_coord, mp→cur_y);
  number_add(z, r1);
  if (number_less(z, h→miny))  number_clone(h→miny, z);
  if (number_greater(z, h→maxy))  number_clone(h→maxy, z);
  free_number(r1);
  free_number(r2);
  free_number(arg1);
}
```

This code is used in section 527.

531. ⟨Advance p to the end of the path and make q the previous knot 531⟩ ≡

```
do
{
  q = p;
  p = mp_next_knot(p);
}
while (mp_right_type(p) ≠ mp_endpoint)
```

This code is used in section 527.

532.   The major difficulty in finding the bounding box of an edge structure is the effect of clipping paths. We treat them conservatively by only clipping to the clipping path's bounding box, but this still requires recursive calls to *set_bbox* in order to find the bounding box of the objects to be clipped. Such calls are distinguished by the fact that the boolean parameter *top_level* is false.

```
void mp_set_bbox(MP mp, mp_edge_header_node h, boolean top_level)
{
  mp_node p;      /* a graphical object being considered */
  integer lev;      /* nesting level for mp_start_bounds_node nodes */      /* Wipe out any existing
      bounding box information if bbtype(h) is incompatible with internal[mp_true_corners] */
  switch (h→bbtype) {
  case no_bounds: break;
  case bounds_set:
    if (number_positive(internal_value(mp_true_corners))) mp_init_bbox(mp, h);
    break;
  case bounds_unset:
    if (number_nonpositive(internal_value(mp_true_corners))) mp_init_bbox(mp, h);
    break;
  }      /* there are no other cases */
  while (mp_link(bblast(h)) ≠ Λ) {
    p = mp_link(bblast(h));
    bblast(h) = p;
    switch (mp_type(p)) {
    case mp_stop_clip_node_type:
      if (top_level) mp_confusion(mp, "bbox");
      else return;
      ;
      break;
    ⟨Other cases for updating the bounding box based on the type of object p  534⟩;
    default:      /* there are no other valid cases, but please the compiler */
      break;
    }
  }
  if (¬top_level) mp_confusion(mp, "bbox");
}
```

533.   ⟨Declarations 8⟩ +≡

```
static void mp_set_bbox(MP mp, mp_edge_header_node h, boolean top_level);
```

534.   ⟨Other cases for updating the bounding box based on the type of object p 534⟩ ≡
case $mp\_fill\_node\_type$: $mp\_path\_bbox(mp, mp\_path\_p((\textbf{mp\_fill\_node})\ p))$;
  if $(mp\_pen\_p((\textbf{mp\_fill\_node})\ p) \neq \Lambda)$ {
    mp_number $x0a$, $y0a$, $x1a$, $y1a$;

    $new\_number(x0a)$;
    $new\_number(y0a)$;
    $new\_number(x1a)$;
    $new\_number(y1a)$;
    $number\_clone(x0a, mp\_minx)$;
    $number\_clone(y0a, mp\_miny)$;
    $number\_clone(x1a, mp\_maxx)$;
    $number\_clone(y1a, mp\_maxy)$;
    $mp\_pen\_bbox(mp, mp\_pen\_p((\textbf{mp\_fill\_node})\ p))$;
    $number\_add(mp\_minx, x0a)$;
    $number\_add(mp\_miny, y0a)$;
    $number\_add(mp\_maxx, x1a)$;
    $number\_add(mp\_maxy, y1a)$;
    $free\_number(x0a)$;
    $free\_number(y0a)$;
    $free\_number(x1a)$;
    $free\_number(y1a)$;
  }
  $mp\_adjust\_bbox(mp, h)$;
  break;

See also sections 535, 537, 538, and 539.

This code is used in section 532.

535.   ⟨Other cases for updating the bounding box based on the type of object p 534⟩ +≡
case $mp\_start\_bounds\_node\_type$:
  if $(number\_positive(internal\_value(mp\_true\_corners)))$ {
    $h{\rightarrow}bbtype = bounds\_unset$;
  }
  else {
    $h{\rightarrow}bbtype = bounds\_set$;
    $mp\_path\_bbox(mp, mp\_path\_p((\textbf{mp\_start\_bounds\_node})\ p))$;
    $mp\_adjust\_bbox(mp, h)$;
    ⟨Scan to the matching **mp_stop_bounds_node** node and update p and $bblast(h)$ 536⟩;
  }
  break;
case $mp\_stop\_bounds\_node\_type$:
  if $(number\_nonpositive(internal\_value(mp\_true\_corners)))$ $mp\_confusion(mp, \texttt{"bbox2"})$;
  ;
  break;

536.   ⟨ Scan to the matching **mp_stop_bounds_node** node and update *p* and *bblast*(*h*) 536 ⟩ ≡
  lev = 1;
  while (*lev* ≠ 0) {
    if (*mp_link*(*p*) ≡ Λ) *mp_confusion*(*mp*, "bbox2");
    ;   /∗ clang: dereference null pointer ∗/
    assert(*mp_link*(*p*));
    p = *mp_link*(*p*);
    if (*mp_type*(*p*) ≡ *mp_start_bounds_node_type*) *incr*(*lev*);
    else if (*mp_type*(*p*) ≡ *mp_stop_bounds_node_type*) *decr*(*lev*);
  }
  bblast(*h*) = *p*

This code is used in section 535.

537.   It saves a lot of grief here to be slightly conservative and not account for omitted parts of dashed
lines. We also don't worry about the material omitted when using butt end caps. The basic computation is
for round end caps and *box_ends* augments it for square end caps.

⟨ Other cases for updating the bounding box based on the type of object *p* 534 ⟩ +≡
case *mp_stroked_node_type*: *mp_path_bbox*(*mp*, *mp_path_p*((**mp_stroked_node**) *p*));
  {
    mp_number *x0a*, *y0a*, *x1a*, *y1a*;
    new_number(*x0a*);
    new_number(*y0a*);
    new_number(*x1a*);
    new_number(*y1a*);
    number_clone(*x0a*, *mp_minx*);
    number_clone(*y0a*, *mp_miny*);
    number_clone(*x1a*, *mp_maxx*);
    number_clone(*y1a*, *mp_maxy*);
    mp_pen_bbox(*mp*, *mp_pen_p*((**mp_stroked_node**) *p*));
    number_add(*mp_minx*, *x0a*);
    number_add(*mp_miny*, *y0a*);
    number_add(*mp_maxx*, *x1a*);
    number_add(*mp_maxy*, *y1a*);
    free_number(*x0a*);
    free_number(*y0a*);
    free_number(*x1a*);
    free_number(*y1a*);
  }
  mp_adjust_bbox(*mp*, *h*);
  if ((*mp_left_type*(*mp_path_p*((**mp_stroked_node**) *p*)) ≡ *mp_endpoint*) ∧ (((**mp_stroked_node**)
        p)→*lcap* ≡ 2))
    mp_box_ends(*mp*, *mp_path_p*((**mp_stroked_node**) *p*), *mp_pen_p*((**mp_stroked_node**) *p*), *h*);
  break;

538.   The height width and depth information stored in a text node determines a rectangle that needs to be transformed according to the transformation parameters stored in the text node.

⟨ Other cases for updating the bounding box based on the type of object p 534 ⟩ +≡

case $mp\_text\_node\_type$:

  {

    mp_number $x0a$, $y0a$, $x1a$, $y1a$, $arg1$;
    mp_text_node $p0 = (\textbf{mp\_text\_node})\ p$;

    $new\_number(x0a)$;
    $new\_number(x1a)$;
    $new\_number(y0a)$;
    $new\_number(y1a)$;
    $new\_number(arg1)$;
    $number\_clone(arg1, p0 \rightarrow depth)$;
    $number\_negate(arg1)$;
    $take\_scaled(x1a, p0 \rightarrow txx, p0 \rightarrow width)$;
    $take\_scaled(y0a, p0 \rightarrow txy, arg1)$;
    $take\_scaled(y1a, p0 \rightarrow txy, p0 \rightarrow height)$;
    $number\_clone(mp\_minx, p0 \rightarrow tx)$;
    $number\_clone(mp\_maxx, mp\_minx)$;
    if $(number\_less(y0a, y1a))$ {
      $number\_add(mp\_minx, y0a)$;
      $number\_add(mp\_maxx, y1a)$;
    }
    else {
      $number\_add(mp\_minx, y1a)$;
      $number\_add(mp\_maxx, y0a)$;
    }
    if $(number\_negative(x1a))\ number\_add(mp\_minx, x1a)$;
    else $number\_add(mp\_maxx, x1a)$;
    $take\_scaled(x1a, p0 \rightarrow tyx, p0 \rightarrow width)$;
    $number\_clone(arg1, p0 \rightarrow depth)$;
    $number\_negate(arg1)$;
    $take\_scaled(y0a, p0 \rightarrow tyy, arg1)$;
    $take\_scaled(y1a, p0 \rightarrow tyy, p0 \rightarrow height)$;
    $number\_clone(mp\_miny, p0 \rightarrow ty)$;
    $number\_clone(mp\_maxy, mp\_miny)$;
    if $(number\_less(y0a, y1a))$ {
      $number\_add(mp\_miny, y0a)$;
      $number\_add(mp\_maxy, y1a)$;
    }
    else {
      $number\_add(mp\_miny, y1a)$;
      $number\_add(mp\_maxy, y0a)$;
    }
    if $(number\_negative(x1a))\ number\_add(mp\_miny, x1a)$;
    else $number\_add(mp\_maxy, x1a)$;
    $mp\_adjust\_bbox(mp, h)$;
    $free\_number(x0a)$;
    $free\_number(y0a)$;
    $free\_number(x1a)$;
    $free\_number(y1a)$;
    $free\_number(arg1)$;

```
    }
  break;
```

539.   This case involves a recursive call that advances *bblast*(*h*) to the node of type **mp_stop_clip_node**
that matches *p*.

⟨ Other cases for updating the bounding box based on the type of object *p* 534 ⟩ +≡
case *mp_start_clip_node_type*:
```
  {
    mp_number sminx, sminy, smaxx, smaxy;
      /∗ for saving the bounding box during recursive calls ∗/
    mp_number x0a, y0a, x1a, y1a;

    new_number(x0a);
    new_number(y0a);
    new_number(x1a);
    new_number(y1a);
    new_number(sminx);
    new_number(sminy);
    new_number(smaxx);
    new_number(smaxy);
    mp_path_bbox(mp, mp_path_p((mp_start_clip_node) p));
    number_clone(x0a, mp_minx);
    number_clone(y0a, mp_miny);
    number_clone(x1a, mp_maxx);
    number_clone(y1a, mp_maxy);
    number_clone(sminx, h⇾minx);
    number_clone(sminy, h⇾miny);
    number_clone(smaxx, h⇾maxx);
    number_clone(smaxy, h⇾maxy);
    ⟨ Reinitialize the bounding box in header h and call set_bbox recursively starting at mp_link(p) 540 ⟩;
    ⟨ Clip the bounding box in h to the rectangle given by x0a, x1a, y0a, y1a 541 ⟩;
    number_clone(mp_minx, sminx);
    number_clone(mp_miny, sminy);
    number_clone(mp_maxx, smaxx);
    number_clone(mp_maxy, smaxy);
    mp_adjust_bbox(mp, h);
    free_number(sminx);
    free_number(sminy);
    free_number(smaxx);
    free_number(smaxy);
    free_number(x0a);
    free_number(y0a);
    free_number(x1a);
    free_number(y1a);
  }
  break;
```

540.   ⟨ Reinitialize the bounding box in header h and call set_bbox recursively starting at
        $mp\_link(p)$ 540 ⟩ ≡
  $set\_number\_to\_inf(h \rightarrow minx);$
  $set\_number\_to\_inf(h \rightarrow miny);$
  $set\_number\_to\_neg\_inf(h \rightarrow maxx);$
  $set\_number\_to\_neg\_inf(h \rightarrow maxy);\ mp\_set\_bbox(mp, h, false)$
This code is used in section 539.

541.   ⟨ Clip the bounding box in h to the rectangle given by $x0a$, $x1a$, $y0a$, $y1a$ 541 ⟩ ≡
  if $(number\_less(h \rightarrow minx, x0a))\ number\_clone(h \rightarrow minx, x0a);$
  if $(number\_less(h \rightarrow miny, y0a))\ number\_clone(h \rightarrow miny, y0a);$
  if $(number\_greater(h \rightarrow maxx, x1a))\ number\_clone(h \rightarrow maxx, x1a);$
  if $(number\_greater(h \rightarrow maxy, y1a))\ number\_clone(h \rightarrow maxy, y1a);$

This code is used in section 539.

542.   Finding an envelope.   When METAPOST has a path and a polygonal pen, it needs to express the desired shape in terms of things PostScript can understand. The present task is to compute a new path that describes the region to be filled. It is convenient to define this as a two step process where the first step is determining what offset to use for each segment of the path.

543.   Given a pointer c to a cyclic path, and a pointer h to the first knot of a pen polygon, the *offset_prep* routine changes the path into cubics that are associated with particular pen offsets. Thus if the cubic between p and q is associated with the kth offset and the cubic between q and r has offset l then $mp\_info(q) = zero\_off + l - k$. (The constant *zero_off* is added to because $l - k$ could be negative.)

   After overwriting the type information with offset differences, we no longer have a true path so we refer to the knot list returned by *offset_prep* as an "envelope spec." Since an envelope spec only determines relative changes in pen offsets, *offset_prep* sets a global variable *spec_offset* to the relative change from h to the first offset.

#define   *zero_off*   16384   /∗ added to offset changes to make them positive ∗/

⟨ Global variables 14 ⟩ +≡
   integer *spec_offset*;   /∗ number of pen edges between h and the initial offset ∗/

544.   **static mp_knot** *mp_offset_prep*(**MP** *mp*, **mp_knot** *c*, **mp_knot** *h*)
  {
    int *n*;   /∗ the number of vertices in the pen polygon ∗/
    mp_knot *c0*, *p*, *q*, *q0*, *r*, *w*, *ww*;   /∗ for list manipulation ∗/
    int *k_needed*;   /∗ amount to be added to *mp_info*(*p*) when it is computed ∗/
    mp_knot *w0*;   /∗ a pointer to pen offset to use just before *p* ∗/
    mp_number *dxin*, *dyin*;   /∗ the direction into knot *p* ∗/
    int *turn_amt*;   /∗ change in pen offsets for the current cubic ∗/
    mp_number *max_coef*;   /∗ used while scaling ∗/
    mp_number *ss*;
    ⟨Other local variables for *offset_prep* 558⟩;
    new_number(*max_coef*);
    new_number(*dxin*);
    new_number(*dyin*);
    new_number(*dx0*);
    new_number(*dy0*);
    new_number(*x0*);
    new_number(*y0*);
    new_number(*x1*);
    new_number(*y1*);
    new_number(*x2*);
    new_number(*y2*);
    new_number(*du*);
    new_number(*dv*);
    new_number(*dx*);
    new_number(*dy*);
    new_number(*x0a*);
    new_number(*y0a*);
    new_number(*x1a*);
    new_number(*y1a*);
    new_number(*x2a*);
    new_number(*y2a*);
    new_number(*t0*);
    new_number(*t1*);
    new_number(*t2*);
    new_number(*u0*);
    new_number(*u1*);
    new_number(*v0*);
    new_number(*v1*);
    new_fraction(*ss*);
    new_fraction(*s*);
    new_fraction(*t*);
    ⟨Initialize the pen size *n* 547⟩;
    ⟨Initialize the incoming direction and pen offset at *c* 548⟩;
    p = *c*;
    c0 = *c*;
    k_needed = 0;
    do {
      q = *mp_next_knot*(*p*);
      ⟨Split the cubic between *p* and *q*, if necessary, into cubics associated with single offsets, after which *q*
          should point to the end of the final such cubic 555⟩;

NOT_FOUND: ⟨Advance p to node q, removing any "dead" cubics that might have been introduced by the splitting process 549⟩;
} **while** $(q \neq c)$;
⟨Fix the offset change in $mp\_knot\_info(c)$ and set c to the return value of $offset\_prep$ 569⟩;
$free\_number(ss)$;
$free\_number(s)$;
$free\_number(dxin)$;
$free\_number(dyin)$;
$free\_number(dx0)$;
$free\_number(dy0)$;
$free\_number(x0)$;
$free\_number(y0)$;
$free\_number(x1)$;
$free\_number(y1)$;
$free\_number(x2)$;
$free\_number(y2)$;
$free\_number(max\_coef)$;
$free\_number(du)$;
$free\_number(dv)$;
$free\_number(dx)$;
$free\_number(dy)$;
$free\_number(x0a)$;
$free\_number(y0a)$;
$free\_number(x1a)$;
$free\_number(y1a)$;
$free\_number(x2a)$;
$free\_number(y2a)$;
$free\_number(t0)$;
$free\_number(t1)$;
$free\_number(t2)$;
$free\_number(u0)$;
$free\_number(u1)$;
$free\_number(v0)$;
$free\_number(v1)$;
$free\_number(t)$;
return c;
}

545.   We shall want to keep track of where certain knots on the cyclic path wind up in the envelope spec. It doesn't suffice just to keep pointers to knot nodes because some nodes are deleted while removing dead cubics. Thus $offset\_prep$ updates the following pointers

⟨Global variables 14⟩ +≡
  mp_knot $spec\_p1$;
  mp_knot $spec\_p2$;   /* pointers to distinguished knots */

546.   ⟨Set initial values of key variables 38⟩ +≡
  $mp{\rightarrow}spec\_p1 = \Lambda$;
  $mp{\rightarrow}spec\_p2 = \Lambda$;

547.   ⟨Initialize the pen size n 547⟩ ≡
  $n = 0$;
  $p = h$; **do**
  {
    $incr(n)$;
    $p = mp\_next\_knot(p)$;
  }
  while $(p \neq h)$

This code is used in section 544.

548.   Since the true incoming direction isn't known yet, we just pick a direction consistent with the pen
offset h. If this is wrong, it can be corrected later.

⟨Initialize the incoming direction and pen offset at c 548⟩ ≡
  {
    mp_knot $hn = mp\_next\_knot(h)$;
    mp_knot $hp = mp\_prev\_knot(h)$;

    $set\_number\_from\_substraction(dxin, hn\text{→}x\_coord, hp\text{→}x\_coord)$;
    $set\_number\_from\_substraction(dyin, hn\text{→}y\_coord, hp\text{→}y\_coord)$;
    if $(number\_zero(dxin) \wedge number\_zero(dyin))$ {
      $set\_number\_from\_substraction(dxin, hp\text{→}y\_coord, h\text{→}y\_coord)$;
      $set\_number\_from\_substraction(dyin, h\text{→}x\_coord, hp\text{→}x\_coord)$;
    }
  }
  $w0 = h$

This code is used in section 544.

549.   We must be careful not to remove the only cubic in a cycle.

But we must also be careful for another reason. If the user-supplied path starts with a set of degenerate
cubics, the target node q can be collapsed to the initial node p which might be the same as the initial node
c of the curve. This would cause the *offset_prep* routine to bail out too early, causing distress later on. (See
for example the testcase reported by Bogusław Jackowski in tracker id 267, case 52c on Sarovar.)

⟨Advance p to node q, removing any "dead" cubics that might have been introduced by the splitting
      process 549⟩ ≡
  $q0 = q$;
  do {
    $r = mp\_next\_knot(p)$;
    if $(number\_equal(p\text{→}x\_coord, p\text{→}right\_x) \wedge number\_equal(p\text{→}y\_coord, p\text{→}right\_y) \wedge number\_equal(p\text{→}x\_coord,$
        $r\text{→}left\_x) \wedge number\_equal(p\text{→}y\_coord, r\text{→}left\_y) \wedge number\_equal(p\text{→}x\_coord,$
        $r\text{→}x\_coord) \wedge number\_equal(p\text{→}y\_coord, r\text{→}y\_coord) \wedge r \neq p \wedge r \neq q)$ {
      ⟨Remove the cubic following p and update the data structures to merge r into p 550⟩;
    }
    $p = r$;
  } **while** $(p \neq q)$;   /* Check if we removed too much */
  if $((q \neq q0) \wedge (q \neq c \vee c \equiv c0))$ $q = mp\_next\_knot(q)$

This code is used in section 544.

550.   ⟨Remove the cubic following p and update the data structures to merge r into p 550⟩ ≡

```
{
    k_needed = mp_knot_info(p) − zero_off;
    if (r ≡ q) {
        q = p;
    }
    else {
        mp_knot_info(p) = k_needed + mp_knot_info(r);
        k_needed = 0;
    }
    if (r ≡ c) {
        mp_knot_info(p) = mp_knot_info(c);
        c = p;
    }
    if (r ≡ mp→spec_p1)  mp→spec_p1 = p;
    if (r ≡ mp→spec_p2)  mp→spec_p2 = p;
    r = p;
    mp_remove_cubic(mp, p);
}
```

This code is used in section 549.

551.   Not setting the *info* field of the newly created knot allows the splitting routine to work for paths.

⟨Declarations 8⟩ +≡

```
static void mp_split_cubic(MP mp, mp_knot p, mp_number t);
```

552.   **void** *mp_split_cubic*(**MP** *mp*, **mp_knot** *p*, **mp_number** *t*)

```
{    /* splits the cubic after p */
    mp_number v;       /* an intermediate value */
    mp_knot q, r;      /* for list manipulation */
    q = mp_next_knot(p);
    r = mp_new_knot(mp);
    mp_next_knot(p) = r;
    mp_next_knot(r) = q;
    mp_originator(r) = mp_program_code;
    mp_left_type(r) = mp_explicit;
    mp_right_type(r) = mp_explicit;
    new_number(v);
    set_number_from_of_the_way(v, t, p→right_x, q→left_x);
    set_number_from_of_the_way(p→right_x, t, p→x_coord, p→right_x);
    set_number_from_of_the_way(q→left_x, t, q→left_x, q→x_coord);
    set_number_from_of_the_way(r→left_x, t, p→right_x, v);
    set_number_from_of_the_way(r→right_x, t, v, q→left_x);
    set_number_from_of_the_way(r→x_coord, t, r→left_x, r→right_x);
    set_number_from_of_the_way(v, t, p→right_y, q→left_y);
    set_number_from_of_the_way(p→right_y, t, p→y_coord, p→right_y);
    set_number_from_of_the_way(q→left_y, t, q→left_y, q→y_coord);
    set_number_from_of_the_way(r→left_y, t, p→right_y, v);
    set_number_from_of_the_way(r→right_y, t, v, q→left_y);
    set_number_from_of_the_way(r→y_coord, t, r→left_y, r→right_y);
    free_number(v);
}
```

553.   This does not set $mp\_knot\_info(p)$ or $mp\_right\_type(p)$.

⟨ Declarations 8 ⟩ +≡
  static void $mp\_remove\_cubic(\textbf{MP } mp, \textbf{mp\_knot } p)$;

554.   **void** $mp\_remove\_cubic(\textbf{MP } mp, \textbf{mp\_knot } p)$
  {   /∗ removes the dead cubic following p ∗/
    mp_knot q;   /∗ the node that disappears ∗/
    (**void**) mp;
    $q = mp\_next\_knot(p)$;
    $mp\_next\_knot(p) = mp\_next\_knot(q)$;
    $number\_clone(p\text{-}right\_x, q\text{-}right\_x)$;
    $number\_clone(p\text{-}right\_y, q\text{-}right\_y)$;
    $mp\_xfree(q)$;
  }

555.   Let $d \prec d'$ mean that the counter-clockwise angle from d to d' is strictly between zero and $180°$. Then we can define $d \preceq d'$ to mean that the angle could be zero or $180°$. If $w_k = (u_k, v_k)$ is the kth pen offset, the kth pen edge direction is defined by the formula

$$d_k = (u_{k+1} - u_k, \, v_{k+1} - v_k).$$

When listed by increasing k, these directions occur in counter-clockwise order so that $d_k \preceq d_{k+1}$ for all k. The goal of *offset_prep* is to find an offset index k to associate with each cubic, such that the direction $d(t)$ of the cubic satisfies

$$d_{k-1} \preceq d(t) \preceq d_k \qquad \text{for } 0 \le t \le 1. \tag{∗}$$

We may have to split a cubic into many pieces before each piece corresponds to a unique offset.

⟨ Split the cubic between p and q, if necessary, into cubics associated with single offsets, after which q should
      point to the end of the final such cubic 555 ⟩ ≡
  $mp\_knot\_info(p) = zero\_off + k\_needed$;
  $k\_needed = 0$;
  ⟨ Prepare for derivative computations; **goto** *not_found* if the current cubic is dead 559 ⟩;
  ⟨ Find the initial direction (dx, dy) 564 ⟩;
  ⟨ Update $mp\_knot\_info(p)$ and find the offset w_k such that $d_{k-1} \preceq (dx, dy) \prec d_k$; also advance $w0$ for the
      direction change at p 566 ⟩;
  ⟨ Find the final direction $(dxin, dyin)$ 565 ⟩;
  ⟨ Decide on the net change in pen offsets and set *turn_amt* 574 ⟩;
  ⟨ Complete the offset splitting process 570 ⟩;
  $w0 = mp\_pen\_walk(mp, w0, turn\_amt)$
This code is used in section 544.

556.   ⟨ Declarations 8 ⟩ +≡
  static mp_knot $mp\_pen\_walk(\textbf{MP } mp, \textbf{mp\_knot } w, \textbf{integer } k)$;

557.   **mp_knot** $mp\_pen\_walk(\mathbf{MP}\ mp, \mathbf{mp\_knot}\ w, \mathbf{integer}\ k)$
 { /∗ walk k steps around a pen from w ∗/
  (**void**) mp;
  while $(k > 0)$ {
   $w = mp\_next\_knot(w)$;
   $decr(k)$;
  }
  while $(k < 0)$ {
   $w = mp\_prev\_knot(w)$;
   $incr(k)$;
  }
  return w;
 }

558.   The direction of a cubic $B(z_0, z_1, z_2, z_3; t) = \big(x(t), y(t)\big)$ can be calculated from the quadratic polynomials $\frac{1}{3}x'(t) = B(x_1 - x_0, x_2 - x_1, x_3 - x_2; t)$ and $\frac{1}{3}y'(t) = B(y_1 - y_0, y_2 - y_1, y_3 - y_2; t)$. Since we may be calculating directions from several cubics split from the current one, it is desirable to do these calculations without losing too much precision. "Scaled up" values of the derivatives, which will be less tainted by accumulated errors than derivatives found from the cubics themselves, are maintained in local variables $x0$, $x1$, and $x2$, representing $X_0 = 2^l(x_1 - x_0)$, $X_1 = 2^l(x_2 - x_1)$, and $X_2 = 2^l(x_3 - x_2)$; similarly $y0$, $y1$, and $y2$ represent $Y_0 = 2^l(y_1 - y_0)$, $Y_1 = 2^l(y_2 - y_1)$, and $Y_2 = 2^l(y_3 - y_2)$.

⟨ Other local variables for *offset_prep* 558 ⟩ ≡
 mp_number $x0$, $x1$, $x2$, $y0$, $y1$, $y2$; /∗ representatives of derivatives ∗/
 mp_number $t0$, $t1$, $t2$; /∗ coefficients of polynomial for slope testing ∗/
 mp_number du, dv, dx, dy; /∗ for directions of the pen and the curve ∗/
 mp_number $dx0$, $dy0$; /∗ initial direction for the first cubic in the curve ∗/
 mp_number $x0a$, $x1a$, $x2a$, $y0a$, $y1a$, $y2a$; /∗ intermediate values ∗/
 mp_number t; /∗ where the derivative passes through zero ∗/
 mp_number s; /∗ a temporary value ∗/

See also section 573.

This code is used in section 544.

559.   ⟨Prepare for derivative computations; **goto** *not_found* if the current cubic is dead 559⟩ ≡
  set_number_from_substraction(*x0*, *p*⇀*right_x*, *p*⇀*x_coord*);
  set_number_from_substraction(*x2*, *q*⇀*x_coord*, *q*⇀*left_x*);
  set_number_from_substraction(*x1*, *q*⇀*left_x*, *p*⇀*right_x*);
  set_number_from_substraction(*y0*, *p*⇀*right_y*, *p*⇀*y_coord*);
  set_number_from_substraction(*y2*, *q*⇀*y_coord*, *q*⇀*left_y*);
  set_number_from_substraction(*y1*, *q*⇀*left_y*, *p*⇀*right_y*);
  {
    mp_number *absval*;

    new_number(*absval*);
    number_clone(*absval*, *x1*);
    number_abs(*absval*);
    number_clone(*max_coef*, *x0*);
    number_abs(*max_coef*);
    if (*number_greater*(*absval*, *max_coef*)) {
      number_clone(*max_coef*, *absval*);
    }
    number_clone(*absval*, *x2*);
    number_abs(*absval*);
    if (*number_greater*(*absval*, *max_coef*)) {
      number_clone(*max_coef*, *absval*);
    }
    number_clone(*absval*, *y0*);
    number_abs(*absval*);
    if (*number_greater*(*absval*, *max_coef*)) {
      number_clone(*max_coef*, *absval*);
    }
    number_clone(*absval*, *y1*);
    number_abs(*absval*);
    if (*number_greater*(*absval*, *max_coef*)) {
      number_clone(*max_coef*, *absval*);
    }
    number_clone(*absval*, *y2*);
    number_abs(*absval*);
    if (*number_greater*(*absval*, *max_coef*)) {
      number_clone(*max_coef*, *absval*);
    }
    if (*number_zero*(*max_coef*)) {
      goto NOT_FOUND;
    }
    free_number(*absval*);
  }
  while (*number_less*(*max_coef*, *fraction_half_t*)) {
    number_double(*max_coef*);
    number_double(*x0*);
    number_double(*x1*);
    number_double(*x2*);
    number_double(*y0*);
    number_double(*y1*);
    number_double(*y2*);
  }

This code is used in section 555.

560.   Let us first solve a special case of the problem: Suppose we know an index k such that either (i) $d(t) \succeq d_{k-1}$ for all t and $d(0) \prec d_k$, or (ii) $d(t) \preceq d_k$ for all t and $d(0) \succ d_{k-1}$. Then, in a sense, we're halfway done, since one of the two relations in $(*)$ is satisfied, and the other couldn't be satisfied for any other value of k.

Actually, the conditions can be relaxed somewhat since a relation such as $d(t) \succeq d_{k-1}$ restricts $d(t)$ to a half plane when all that really matters is whether $d(t)$ crosses the ray in the d_{k-1} direction from the origin. The condition for case (i) becomes $d_{k-1} \preceq d(0) \prec d_k$ and $d(t)$ never crosses the d_{k-1} ray in the clockwise direction. Case (ii) is similar except $d(t)$ cannot cross the d_k ray in the counterclockwise direction.

The *fin_offset_prep* subroutine solves the stated subproblem. It has a parameter called *rise* that is 1 in case (i), -1 in case (ii). Parameters *x0* through *y2* represent the derivative of the cubic following p. The w parameter should point to offset w_k and $mp\_info(p)$ should already be set properly. The *turn_amt* parameter gives the absolute value of the overall net change in pen offsets.

⟨ Declarations 8 ⟩ +≡
  static void *mp_fin_offset_prep* (**MP** *mp*, **mp_knot** *p*, **mp_knot** *w*, **mp_number** *x0*, **mp_number**
      x1, **mp_number** *x2*, **mp_number** *y0*, **mp_number** *y1*, **mp_number** *y2*, **integer** *rise*, **integer**
      turn_amt);

561.   **void** *mp_fin_offset_prep*(**MP** *mp*, **mp_knot** *p*, **mp_knot** *w*, **mp_number** *x0*, **mp_number**
           x1, **mp_number** *x2*, **mp_number** *y0*, **mp_number** *y1*, **mp_number** *y2*, **integer**
           rise, **integer** *turn_amt*)
{
   mp_knot *ww*;   /∗ for list manipulation ∗/
   mp_number *du*, *dv*;   /∗ for slope calculation ∗/
   mp_number *t0*, *t1*, *t2*;   /∗ test coefficients ∗/
   mp_number *t*;   /∗ place where the derivative passes a critical slope ∗/
   mp_number *s*;   /∗ slope or reciprocal slope ∗/
   mp_number *v*;   /∗ intermediate value for updating *x0* .. *y2* ∗/
   mp_knot *q*;   /∗ original *mp_next_knot*(*p*) ∗/
   q = *mp_next_knot*(*p*);
   new_number(*du*);
   new_number(*dv*);
   new_number(*v*);
   new_number(*t0*);
   new_number(*t1*);
   new_number(*t2*);
   new_fraction(*s*);
   new_fraction(*t*);
   while (1) {
      if (*rise* > 0) *ww* = *mp_next_knot*(*w*);   /∗ a pointer to w_{k+1} ∗/
      else *ww* = *mp_prev_knot*(*w*);   /∗ a pointer to w_{k-1} ∗/
      ⟨Compute test coefficients (*t0*, *t1*, *t2*) for *d*(*t*) versus d_k or d_{k-1} 562⟩;
      crossing_point(*t*, *t0*, *t1*, *t2*);
      if (*number_greaterequal*(*t*, *fraction_one_t*)) {
         if (*turn_amt* > 0) *number_clone*(*t*, *fraction_one_t*);
         else goto RETURN;
      }
      ⟨Split the cubic at *t*, and split off another cubic if the derivative crosses back 563⟩;
      w = *ww*;
   }
RETURN: *free_number*(*s*);
   free_number(*t*);
   free_number(*du*);
   free_number(*dv*);
   free_number(*v*);
   free_number(*t0*);
   free_number(*t1*);
   free_number(*t2*);
}

562.   We want $B(t0, t1, t2; t)$ to be the dot product of $d(t)$ with a $-90°$ rotation of the vector from w to ww. This makes the resulting function cross from positive to negative when $d_{k-1} \preceq d(t) \preceq d_k$ begins to fail.

$\langle$ Compute test coefficients $(t0, t1, t2)$ for $d(t)$ versus d_k or d_{k-1} 562 $\rangle \equiv$

 {

  mp_number *abs_du*, *abs_dv*;

  new_number(*abs_du*);
  new_number(*abs_dv*);
  set_number_from_substraction(*du*, *ww*→*x_coord*, *w*→*x_coord*);
  set_number_from_substraction(*dv*, *ww*→*y_coord*, *w*→*y_coord*);
  number_clone(*abs_du*, *du*);
  number_abs(*abs_du*);
  number_clone(*abs_dv*, *dv*);
  number_abs(*abs_dv*);
  if (*number_greaterequal*(*abs_du*, *abs_dv*)) {
   mp_number *r1*;

   new_fraction(*r1*);
   make_fraction(*s*, *dv*, *du*);
   take_fraction(*r1*, *x0*, *s*);
   set_number_from_substraction(*t0*, *r1*, *y0*);
   take_fraction(*r1*, *x1*, *s*);
   set_number_from_substraction(*t1*, *r1*, *y1*);
   take_fraction(*r1*, *x2*, *s*);
   set_number_from_substraction(*t2*, *r1*, *y2*);
   if (*number_negative*(*du*)) {
    number_negate(*t0*);
    number_negate(*t1*);
    number_negate(*t2*);
   }
   free_number(*r1*);
  }
  else {
   mp_number *r1*;

   new_fraction(*r1*);
   make_fraction(*s*, *du*, *dv*);
   take_fraction(*r1*, *y0*, *s*);
   set_number_from_substraction(*t0*, *x0*, *r1*);
   take_fraction(*r1*, *y1*, *s*);
   set_number_from_substraction(*t1*, *x1*, *r1*);
   take_fraction(*r1*, *y2*, *s*);
   set_number_from_substraction(*t2*, *x2*, *r1*);
   if (*number_negative*(*dv*)) {
    number_negate(*t0*);
    number_negate(*t1*);
    number_negate(*t2*);
   }
   free_number(*r1*);
  }
  free_number(*abs_du*);
  free_number(*abs_dv*);
  if (*number_negative*(*t0*)) *set_number_to_zero*(*t0*); /* should be positive without rounding error */
 }

This code is used in sections 561 and 570.

563.   The curve has crossed d_k or d_{k-1}; its initial segment satisfies $(*)$, and it might cross again and return towards s_{k-1} or s_k, respectively, yielding another solution of $(*)$.

$\langle$ Split the cubic at t, and split off another cubic if the derivative crosses back $563\,\rangle \equiv$

```
{
  mp_split_cubic(mp, p, t);
  p = mp_next_knot(p);
  mp_knot_info(p) = zero_off + rise;
  decr(turn_amt);
  set_number_from_of_the_way(v, t, x0, x1);
  set_number_from_of_the_way(x1, t, x1, x2);
  set_number_from_of_the_way(x0, t, v, x1);
  set_number_from_of_the_way(v, t, y0, y1);
  set_number_from_of_the_way(y1, t, y1, y2);
  set_number_from_of_the_way(y0, t, v, y1);
  if (turn_amt < 0) {
    mp_number arg1, arg2, arg3;

    new_number(arg1);
    new_number(arg2);
    new_number(arg3);
    set_number_from_of_the_way(t1, t, t1, t2);
    if (number_positive(t1)) set_number_to_zero(t1);      /* without rounding error, t1 would be ≤ 0 */
    number_clone(arg2, t1);
    number_negate(arg2);
    number_clone(arg3, t2);
    number_negate(arg3);
    crossing_point(t, arg1, arg2, arg3);
    free_number(arg1);
    free_number(arg2);
    free_number(arg3);
    if (number_greater(t, fraction_one_t)) number_clone(t, fraction_one_t);
    incr(turn_amt);
    if (number_equal(t, fraction_one_t) ∧ (mp_next_knot(p) ≠ q)) {
      mp_knot_info(mp_next_knot(p)) = mp_knot_info(mp_next_knot(p)) − rise;
    }
  }
  else {
    mp_split_cubic(mp, p, t);
    mp_knot_info(mp_next_knot(p)) = zero_off − rise;
    set_number_from_of_the_way(v, t, x1, x2);
    set_number_from_of_the_way(x1, t, x0, x1);
    set_number_from_of_the_way(x2, t, x1, v);
    set_number_from_of_the_way(v, t, y1, y2);
    set_number_from_of_the_way(y1, t, y0, y1);
    set_number_from_of_the_way(y2, t, y1, v);
  }
}
```

This code is used in section 561.

564.   Now we must consider the general problem of *offset_prep*, when nothing is known about a given cubic. We start by finding its direction in the vicinity of $t = 0$.

If $z'(t) = 0$, the given cubic is numerically unstable but *offset_prep* has not yet introduced any more numerical errors. Thus we can compute the true initial direction for the given cubic, even if it is almost degenerate.

⟨ Find the initial direction (dx, dy) 564 ⟩ ≡
  number_clone(*dx*, *x0*);
  number_clone(*dy*, *y0*);
  if (*number_zero*(*dx*) ∧ *number_zero*(*dy*)) {
    number_clone(*dx*, *x1*);
    number_clone(*dy*, *y1*);
    if (*number_zero*(*dx*) ∧ *number_zero*(*dy*)) {
      number_clone(*dx*, *x2*);
      number_clone(*dy*, *y2*);
    }
  }
  if ($p ≡ c$) {
    number_clone(*dx0*, *dx*);
    number_clone(*dy0*, *dy*);
  }

This code is used in section 555.

565.   ⟨ Find the final direction $(dxin, dyin)$ 565 ⟩ ≡
  number_clone(*dxin*, *x2*);
  number_clone(*dyin*, *y2*);
  if (*number_zero*(*dxin*) ∧ *number_zero*(*dyin*)) {
    number_clone(*dxin*, *x1*);
    number_clone(*dyin*, *y1*);
    if (*number_zero*(*dxin*) ∧ *number_zero*(*dyin*)) {
      number_clone(*dxin*, *x0*);
      number_clone(*dyin*, *y0*);
    }
  }

This code is used in section 555.

566.   The next step is to bracket the initial direction between consecutive edges of the pen polygon. We must be careful to turn clockwise only if this makes the turn less than $180°$. (A $180°$ turn must be counterclockwise in order to make **doublepath** envelopes come out right.) This code depends on $w0$ being the offset for $(dxin, dyin)$.

⟨ Update $mp\_knot\_info(p)$ and find the offset w_k such that $d_{k-1} \preceq (dx, dy) \prec d_k$; also advance $w0$ for the
direction change at p 566 ⟩ ≡

```
{
   mp_number ab_vs_cd;

   new_number (ab_vs_cd);
   ab_vs_cd (ab_vs_cd, dy, dxin, dx, dyin);
   turn_amt = mp_get_turn_amt(mp, w0, dx, dy, number_nonnegative(ab_vs_cd));
   free_number (ab_vs_cd);
   w = mp_pen_walk(mp, w0, turn_amt);
   w0 = w;
   mp_knot_info(p) = mp_knot_info(p) + turn_amt;
}
```

This code is used in section 555.

567.   Decide how many pen offsets to go away from w in order to find the offset for (dx, dy), going counterclockwise if ccw is $true$. This assumes that w is the offset for some direction (x', y') from which the angle to (dx, dy) in the sense determined by ccw is less than or equal to $180°$.

If the pen polygon has only two edges, they could both be parallel to (dx, dy). In this case, we must be careful to stop after crossing the first such edge in order to avoid an infinite loop.

⟨ Declarations 8 ⟩ +≡

static integer $mp\_get\_turn\_amt(\textbf{MP}\ mp, \textbf{mp\_knot}\ w, \textbf{mp\_number}\ dx, \textbf{mp\_number}\ dy, \textbf{boolean}$
$ccw);$

568.   **integer** $mp\_get\_turn\_amt$(**MP** mp, **mp_knot** w, **mp_number** dx, **mp_number** dy, **boolean** ccw)
```
{
```
  mp_knot ww;   /∗ a neighbor of knot w ∗/
  integer s;   /∗ turn amount so far ∗/
  mp_number t;   /∗ ab_vs_cd result ∗/
  mp_number $arg1$, $arg2$;
  $s = 0$;
  $new\_number(arg1)$;
  $new\_number(arg2)$;
  $new\_number(t)$;
  if (ccw) {
    $ww = mp\_next\_knot(w)$;
    do {
      $set\_number\_from\_substraction(arg1, ww{\rightarrow}x\_coord, w{\rightarrow}x\_coord)$;
      $set\_number\_from\_substraction(arg2, ww{\rightarrow}y\_coord, w{\rightarrow}y\_coord)$;
      $ab\_vs\_cd(t, dy, arg1, dx, arg2)$;
      if ($number\_negative(t)$) **break**;
      $incr(s)$;
      $w = ww$;
      $ww = mp\_next\_knot(ww)$;
    } **while** ($number\_positive(t)$);
  }
  else {
    $ww = mp\_prev\_knot(w)$;
    $set\_number\_from\_substraction(arg1, w{\rightarrow}x\_coord, ww{\rightarrow}x\_coord)$;
    $set\_number\_from\_substraction(arg2, w{\rightarrow}y\_coord, ww{\rightarrow}y\_coord)$;
    $ab\_vs\_cd(t, dy, arg1, dx, arg2)$;
    while ($number\_negative(t)$) {
      $decr(s)$;
      $w = ww$;
      $ww = mp\_prev\_knot(ww)$;
      $set\_number\_from\_substraction(arg1, w{\rightarrow}x\_coord, ww{\rightarrow}x\_coord)$;
      $set\_number\_from\_substraction(arg2, w{\rightarrow}y\_coord, ww{\rightarrow}y\_coord)$;
      $ab\_vs\_cd(t, dy, arg1, dx, arg2)$;
    }
  }
  $free\_number(t)$;
  $free\_number(arg1)$;
  $free\_number(arg2)$;
  return s;
}

569.   When we're all done, the final offset is *w0* and the final curve direction is $(dxin, dyin)$. With this knowledge of the incoming direction at *c*, we can correct $mp\_info(c)$ which was erroneously based on an incoming offset of *h*.

#define $fix\_by(A)$ $mp\_knot\_info(c) = mp\_knot\_info(c) + (A)$

⟨ Fix the offset change in $mp\_knot\_info(c)$ and set *c* to the return value of *offset_prep* 569 ⟩ ≡

   $mp{\rightarrow}spec\_offset = mp\_knot\_info(c) - zero\_off$;

   if $(mp\_next\_knot(c) \equiv c)$ {

     $mp\_knot\_info(c) = zero\_off + n$;

   }

   else {

     mp_number ab_vs_cd;

     $new\_number(ab\_vs\_cd)$;

     $fix\_by(k\_needed)$;

     while $(w0 \neq h)$ {

       $fix\_by(1)$;

       $w0 = mp\_next\_knot(w0)$;

     }

     while $(mp\_knot\_info(c) \leq zero\_off - n)$ $fix\_by(n)$;

     while $(mp\_knot\_info(c) > zero\_off)$ $fix\_by(-n)$;

     $ab\_vs\_cd(ab\_vs\_cd, dy0, dxin, dx0, dyin)$;

     if $((mp\_knot\_info(c) \neq zero\_off) \wedge number\_nonnegative(ab\_vs\_cd))$ $fix\_by(n)$;

     $free\_number(ab\_vs\_cd)$;

   }

This code is used in section 544.

570.   Finally we want to reduce the general problem to situations that *fin_offset_prep* can handle. We split
the cubic into at most three parts with respect to d_{k-1}, and apply *fin_offset_prep* to each part.

⟨ Complete the offset splitting process 570 ⟩ ≡

    $ww = mp\_prev\_knot(w);$
    ⟨ Compute test coefficients $(t0, t1, t2)$ for $d(t)$ versus d_k or d_{k-1} 562 ⟩;
    ⟨ Find the first t where $d(t)$ crosses d_{k-1} or set $t: = fraction\_one + 1$ 572 ⟩;
    if $(number\_greater(t, fraction\_one\_t))$ {
        $mp\_fin\_offset\_prep(mp, p, w, x0, x1, x2, y0, y1, y2, 1, turn\_amt);$
    }
    else {
        $mp\_split\_cubic(mp, p, t);$
        $r = mp\_next\_knot(p);$
        $set\_number\_from\_of\_the\_way(x1a, t, x0, x1);$
        $set\_number\_from\_of\_the\_way(x1, t, x1, x2);$
        $set\_number\_from\_of\_the\_way(x2a, t, x1a, x1);$
        $set\_number\_from\_of\_the\_way(y1a, t, y0, y1);$
        $set\_number\_from\_of\_the\_way(y1, t, y1, y2);$
        $set\_number\_from\_of\_the\_way(y2a, t, y1a, y1);$
        $mp\_fin\_offset\_prep(mp, p, w, x0, x1a, x2a, y0, y1a, y2a, 1, 0);$
        $number\_clone(x0, x2a);$
        $number\_clone(y0, y2a);$
        $mp\_knot\_info(r) = zero\_off - 1;$
        if $(turn\_amt \geq 0)$ {
            mp_number $arg1, arg2, arg3;$

            $new\_number(arg1);$
            $new\_number(arg2);$
            $new\_number(arg3);$
            $set\_number\_from\_of\_the\_way(t1, t, t1, t2);$
            if $(number\_positive(t1))$ $set\_number\_to\_zero(t1);$
            $number\_clone(arg2, t1);$
            $number\_negate(arg2);$
            $number\_clone(arg3, t2);$
            $number\_negate(arg3);$
            $crossing\_point(t, arg1, arg2, arg3);$
            $free\_number(arg1);$
            $free\_number(arg2);$
            $free\_number(arg3);$
            if $(number\_greater(t, fraction\_one\_t))$ $number\_clone(t, fraction\_one\_t);$
            ⟨ Split off another rising cubic for *fin_offset_prep* 571 ⟩;
            $mp\_fin\_offset\_prep(mp, r, ww, x0, x1, x2, y0, y1, y2, -1, 0);$
        }
        else {
            $mp\_fin\_offset\_prep(mp, r, ww, x0, x1, x2, y0, y1, y2, -1, (-1 - turn\_amt));$
        }
    }

This code is used in section 555.

571.   ⟨ Split off another rising cubic for *fin_offset_prep* 571 ⟩ ≡

 mp_split_cubic(*mp*, *r*, *t*);
 mp_knot_info(*mp_next_knot*(*r*)) = *zero_off* + 1;
 set_number_from_of_the_way(*x1a*, *t*, *x1*, *x2*);
 set_number_from_of_the_way(*x1*, *t*, *x0*, *x1*);
 set_number_from_of_the_way(*x0a*, *t*, *x1*, *x1a*);
 set_number_from_of_the_way(*y1a*, *t*, *y1*, *y2*);
 set_number_from_of_the_way(*y1*, *t*, *y0*, *y1*);
 set_number_from_of_the_way(*y0a*, *t*, *y1*, *y1a*);
 mp_fin_offset_prep(*mp*, *mp_next_knot*(*r*), *w*, *x0a*, *x1a*, *x2*, *y0a*, *y1a*, *y2*, 1, *turn_amt*);
 number_clone(*x2*, *x0a*); *number_clone*(*y2*, *y0a*)

This code is used in section 570.

572.   At this point, the direction of the incoming pen edge is $(-du, -dv)$. When the component of $d(t)$ perpendicular to $(-du, -dv)$ crosses zero, we need to decide whether the directions are parallel or antiparallel. We can test this by finding the dot product of $d(t)$ and $(-du, -dv)$, but this should be avoided when the value of *turn_amt* already determines the answer. If $t2 < 0$, there is one crossing and it is antiparallel only if $turn\_amt \geq 0$. If $turn\_amt < 0$, there should always be at least one crossing and the first crossing cannot be antiparallel.

$\langle$ Find the first t where $d(t)$ crosses d_{k-1} or set $t: = fraction\_one + 1$ 572 $\rangle \equiv$

```
crossing_point(t, t0, t1, t2);
if (turn_amt ≥ 0) {
  if (number_negative(t2)) {
    number_clone(t, fraction_one_t);
    number_add_scaled(t, 1);
  }
  else {
    mp_number tmp, arg1, r1;

    new_fraction(r1);
    new_number(tmp);
    new_number(arg1);
    set_number_from_of_the_way(u0, t, x0, x1);
    set_number_from_of_the_way(u1, t, x1, x2);
    set_number_from_of_the_way(tmp, t, u0, u1);
    number_clone(arg1, du);
    number_abs(arg1);
    take_fraction(ss, arg1, tmp);
    set_number_from_of_the_way(v0, t, y0, y1);
    set_number_from_of_the_way(v1, t, y1, y2);
    set_number_from_of_the_way(tmp, t, v0, v1);
    number_clone(arg1, dv);
    number_abs(arg1);
    take_fraction(r1, arg1, tmp);
    number_add(ss, r1);
    free_number(tmp);
    if (number_negative(ss)) {
      number_clone(t, fraction_one_t);
      number_add_scaled(t, 1);
    }
    free_number(arg1);
    free_number(r1);
  }
}
else if (number_greater(t, fraction_one_t)) {
  number_clone(t, fraction_one_t);
}
```

This code is used in section 570.

573. $\langle$ Other local variables for *offset_prep* 558 $\rangle$ $+\equiv$
mp_number $u0$, $u1$, $v0$, $v1$;   /* intermediate values for $d(t)$ calculation */
int d_sign;   /* sign of overall change in direction for this cubic */

574.   If the cubic almost has a cusp, it is a numerically ill-conditioned problem to decide which way it loops around but that's OK as long we're consistent. To make **doublepath** envelopes work properly, reversing the path should always change the sign of $turn\_amt$.

⟨ Decide on the net change in pen offsets and set $turn\_amt$ 574 ⟩ ≡
```
  {
    mp_number ab_vs_cd;

    new_number(ab_vs_cd);
    ab_vs_cd(ab_vs_cd, dx, dyin, dxin, dy);
    if (number_negative(ab_vs_cd)) d_sign = −1;
    else if (number_zero(ab_vs_cd)) d_sign = 0;
    else d_sign = 1;
    free_number(ab_vs_cd);
  }
  if (d_sign ≡ 0) {⟨ Check rotation direction based on node position 575 ⟩}
  if (d_sign ≡ 0) {
    if (number_zero(dx)) {
      if (number_positive(dy)) d_sign = 1;
      else d_sign = −1;
    }
    else {
      if (number_positive(dx)) d_sign = 1;
      else d_sign = −1;
    }
  }
⟨ Make ss negative if and only if the total change in direction is more than 180° 576 ⟩;
  turn_amt = mp_get_turn_amt(mp, w, dxin, dyin, (d_sign > 0)); if (number_negative(ss))
      turn_amt = turn_amt − d_sign * n
```
This code is used in section 555.

575.   We check rotation direction by looking at the vector connecting the current node with the next. If its angle with incoming and outgoing tangents has the same sign, we pick this as *d_sign*, since it means we have a flex, not a cusp. Otherwise we proceed to the cusp code.

⟨ Check rotation direction based on node position 575 ⟩ ≡
```
  {
    mp_number ab_vs_cd1, ab_vs_cd2, t;
    new_number(ab_vs_cd1);
    new_number(ab_vs_cd2);
    new_number(t);
    set_number_from_substraction(u0, q→x_coord, p→x_coord);
    set_number_from_substraction(u1, q→y_coord, p→y_coord);
    ab_vs_cd(ab_vs_cd1, dx, u1, u0, dy);
    ab_vs_cd(ab_vs_cd2, u0, dyin, dxin, u1);
    set_number_from_addition(t, ab_vs_cd1, ab_vs_cd2);
    number_half(t);
    if (number_negative(t))  d_sign = −1;
    else if (number_zero(t))  d_sign = 0;
    else  d_sign = 1;
    free_number(t);
    free_number(ab_vs_cd1);
    free_number(ab_vs_cd2);
  }
```
This code is used in section 574.

576.   In order to be invariant under path reversal, the result of this computation should not change when
$x0$, $y0$, ... are all negated and $(x0, y0)$ is then swapped with $(x2, y2)$. We make use of the identities
take_fraction$(-a, -b) =$ *take_fraction*(a, b) and *t_of_the_way*$(-a, -b) = -($*t_of_the_way*$(a, b))$.

⟨ Make *ss* negative if and only if the total change in direction is more than $180°$ 576 ⟩ ≡
  {
    mp_number *r1*, *r2*, *arg1*;

    new_number(*arg1*);
    new_fraction(*r1*);
    new_fraction(*r2*);
    take_fraction(*r1*, *x0*, *y2*);
    take_fraction(*r2*, *x2*, *y0*);
    number_half(*r1*);
    number_half(*r2*);
    set_number_from_substraction(*t0*, *r1*, *r2*);
    set_number_from_addition(*arg1*, *y0*, *y2*);
    take_fraction(*r1*, *x1*, *arg1*);
    set_number_from_addition(*arg1*, *x0*, *x2*);
    take_fraction(*r1*, *y1*, *arg1*);
    number_half(*r1*);
    number_half(*r2*);
    set_number_from_substraction(*t1*, *r1*, *r2*);
    free_number(*arg1*);
    free_number(*r1*);
    free_number(*r2*);
  }
  if (*number_zero*(*t0*)) *set_number_from_scaled*(*t0*, *d_sign*);   /* path reversal always negates *d_sign* */
  if (*number_positive*(*t0*)) {
    mp_number *arg3*;

    new_number(*arg3*);
    number_clone(*arg3*, *t0*);
    number_negate(*arg3*);
    crossing_point(*t*, *t0*, *t1*, *arg3*);
    free_number(*arg3*);
    set_number_from_of_the_way(*u0*, *t*, *x0*, *x1*);
    set_number_from_of_the_way(*u1*, *t*, *x1*, *x2*);
    set_number_from_of_the_way(*v0*, *t*, *y0*, *y1*);
    set_number_from_of_the_way(*v1*, *t*, *y1*, *y2*);
  }
  else {
    mp_number *arg1*;

    new_number(*arg1*);
    number_clone(*arg1*, *t0*);
    number_negate(*arg1*);
    crossing_point(*t*, *arg1*, *t1*, *t0*);
    free_number(*arg1*);
    set_number_from_of_the_way(*u0*, *t*, *x2*, *x1*);
    set_number_from_of_the_way(*u1*, *t*, *x1*, *x0*);
    set_number_from_of_the_way(*v0*, *t*, *y2*, *y1*);
    set_number_from_of_the_way(*v1*, *t*, *y1*, *y0*);
  }
  {

　　　mp_number $tmp1$, $tmp2$, $r1$, $r2$, $arg1$;

　　　$new\_fraction(r1)$;
　　　$new\_fraction(r2)$;
　　　$new\_number(arg1)$;
　　　$new\_number(tmp1)$;
　　　$new\_number(tmp2)$;
　　　$set\_number\_from\_of\_the\_way(tmp1, t, u0, u1)$;
　　　$set\_number\_from\_of\_the\_way(tmp2, t, v0, v1)$;
　　　$set\_number\_from\_addition(arg1, x0, x2)$;
　　　$take\_fraction(r1, arg1, tmp1)$;
　　　$set\_number\_from\_addition(arg1, y0, y2)$;
　　　$take\_fraction(r2, arg1, tmp2)$;
　　　$set\_number\_from\_addition(ss, r1, r2)$;
　　　$free\_number(arg1)$;
　　　$free\_number(r1)$;
　　　$free\_number(r2)$;
　　　$free\_number(tmp1)$;
　　　$free\_number(tmp2)$;
　　}

This code is used in section 574.

577.　　Here's a routine that prints an envelope spec in symbolic form. It assumes that the *cur_pen* has not been walked around to the first offset.

　　static void $mp\_print\_spec(\textbf{MP}\ mp, \textbf{mp\_knot}\ cur\_spec, \textbf{mp\_knot}\ cur\_pen, \textbf{const char}\ *s)$
　　{
　　　mp_knot p, q;　　　/∗ list traversal ∗/
　　　mp_knot w;　　/∗ the current pen offset ∗/

　　　$mp\_print\_diagnostic(mp, \texttt{"Envelope}_\sqcup\texttt{spec"}, s, true)$;
　　　$p = cur\_spec$;
　　　$w = mp\_pen\_walk(mp, cur\_pen, mp\text{→}spec\_offset)$;
　　　$mp\_print\_ln(mp)$;
　　　$mp\_print\_two(mp, cur\_spec\text{→}x\_coord, cur\_spec\text{→}y\_coord)$;
　　　$mp\_print(mp, \texttt{"}_\sqcup\texttt{\%}_\sqcup\texttt{beginning}_\sqcup\texttt{with}_\sqcup\texttt{offset}_\sqcup\texttt{"})$;
　　　$mp\_print\_two(mp, w\text{→}x\_coord, w\text{→}y\_coord)$;
　　　do {
　　　　while (1) {
　　　　　$q = mp\_next\_knot(p)$;
　　　　　⟨Print the cubic between p and q 579⟩;
　　　　　$p = q$;
　　　　　if $((p \equiv cur\_spec) \vee (mp\_knot\_info(p) \neq zero\_off))$ **break**;
　　　　}
　　　　if $(mp\_knot\_info(p) \neq zero\_off)$ {
　　　　　⟨Update w as indicated by $mp\_knot\_info(p)$ and print an explanation 578⟩;
　　　　}
　　　} **while** $(p \neq cur\_spec)$;
　　　$mp\_print\_nl(mp, \texttt{"}_\sqcup\texttt{\&}_\sqcup\texttt{cycle"})$;
　　　$mp\_end\_diagnostic(mp, true)$;
　　}

578.   ⟨Update w as indicated by $mp\_knot\_info(p)$ and print an explanation 578⟩ ≡
  {
    $w = mp\_pen\_walk(mp, w, (mp\_knot\_info(p) - zero\_off));$
    $mp\_print(mp, "\_\%\_");$
    if $(mp\_knot\_info(p) > zero\_off)$ $mp\_print(mp, "counter");$
    $mp\_print(mp, "clockwise\_to\_offset\_");$
    $mp\_print\_two(mp, w{\to}x\_coord, w{\to}y\_coord);$
  }

This code is used in section 577.

579.   ⟨Print the cubic between p and q 579⟩ ≡
  {
    $mp\_print\_nl(mp, "\_\_\_..controls\_");$
    $mp\_print\_two(mp, p{\to}right\_x, p{\to}right\_y);$
    $mp\_print(mp, "\_and\_");$
    $mp\_print\_two(mp, q{\to}left\_x, q{\to}left\_y);$
    $mp\_print\_nl(mp, "\_..");$
    $mp\_print\_two(mp, q{\to}x\_coord, q{\to}y\_coord);$
  }

This code is used in section 577.

580.   Once we have an envelope spec, the remaining task to construct the actual envelope by offsetting each cubic as determined by the *info* fields in the knots. First we use *offset_prep* to convert the c into an envelope spec. Then we add the offsets so that c becomes a cyclic path that represents the envelope.

The *ljoin* and *miterlim* parameters control the treatment of points where the pen offset changes, and *lcap* controls the endpoints of a **doublepath**. The endpoints are easily located because c is given in undoubled form and then doubled in this procedure. We use *spec_p1* and *spec_p2* to keep track of the endpoints and treat them like very sharp corners. Butt end caps are treated like beveled joins; round end caps are treated like round joins; and square end caps are achieved by setting *join_type*: = 3.

None of these parameters apply to inside joins where the convolution tracing has retrograde lines. In such cases we use a simple connect-the-endpoints approach that is achieved by setting *join_type*: = 2.

```
static mp_knot  mp_make_envelope(MP  mp, mp_knot  c, mp_knot  h, quarterword
        ljoin, quarterword  lcap, mp_number  miterlim)
{
    mp_knot p, q, r, q0;       /* for manipulating the path */
    mp_knot w, w0;       /* the pen knot for the current offset */
    halfword k, k0;       /* controls pen edge insertion */
    mp_number qx, qy;       /* unshifted coordinates of q */

    mp_fraction dxin, dyin, dxout, dyout;       /* directions at q when square or mitered */

    int join_type = 0;       /* codes 0..3 for mitered, round, beveled, or square */

    ⟨ Other local variables for make_envelope 584 ⟩;
    new_number(max_ht);
    new_number(tmp);
    new_fraction(dxin);
    new_fraction(dyin);
    new_fraction(dxout);
    new_fraction(dyout);
    mp→spec_p1 = Λ;
    mp→spec_p2 = Λ;
    new_number(qx);
    new_number(qy);
    ⟨ If endpoint, double the path c, and set spec_p1 and spec_p2 595 ⟩;
    ⟨ Use offset_prep to compute the envelope spec then walk h around to the initial offset 581 ⟩;
    w = h;
    p = c;
    do {
        q = mp_next_knot(p);
        q0 = q;
        number_clone(qx, q→x_coord);
        number_clone(qy, q→y_coord);
        k = mp_knot_info(q);
        k0 = k;
        w0 = w;
        if (k ≠ zero_off) {
            ⟨ Set join_type to indicate how to handle offset changes at q 582 ⟩;
        }
        ⟨ Add offset w to the cubic from p to q 585 ⟩;
        while (k ≠ zero_off) {
            ⟨ Step w and move k one step closer to zero_off 586 ⟩;
            if ((join_type ≡ 1) ∨ (k ≡ zero_off)) {
                mp_number xtot, ytot;

                new_number(xtot);
```

$new\_number(ytot);$
$set\_number\_from\_addition(xtot, qx, w\text{→}x\_coord);$
$set\_number\_from\_addition(ytot, qy, w\text{→}y\_coord);$
$q = mp\_insert\_knot(mp, q, xtot, ytot);$
        }
      }
      if $(q \neq mp\_next\_knot(p))$ {
        ⟨Set $p = mp\_link(p)$ and add knots between p and q as required by $join\_type$ 589⟩;
      }
      $p = q;$
    } **while** $(q0 \neq c);$
    $free\_number(max\_ht);$
    $free\_number(tmp);$
    $free\_number(qx);$
    $free\_number(qy);$
    $free\_number(dxin);$
    $free\_number(dyin);$
    $free\_number(dxout);$
    $free\_number(dyout);$
    return $c;$
  }

581.   ⟨Use $offset\_prep$ to compute the envelope spec then walk h around to the initial offset 581⟩ ≡
  $c = mp\_offset\_prep(mp, c, h);$
  if $(number\_positive(internal\_value(mp\_tracing\_specs)))$ $mp\_print\_spec(mp, c, h, "");$
  $h = mp\_pen\_walk(mp, h, mp\text{→}spec\_offset)$

This code is used in section 580.

582.   Mitered and squared-off joins depend on path directions that are difficult to compute for degenerate
cubics. The envelope spec computed by $offset\_prep$ can have degenerate cubics only if the entire cycle
collapses to a single degenerate cubic. Setting $join\_type$: $= 2$ in this case makes the computed envelope
degenerate as well.

⟨Set $join\_type$ to indicate how to handle offset changes at q 582⟩ ≡
  if $(k < zero\_off)$ {
    $join\_type = 2;$
  }
  else {
    if $((q \neq mp\text{→}spec\_p1) \wedge (q \neq mp\text{→}spec\_p2))$ $join\_type = ljoin;$
    else if $(lcap \equiv 2)$ $join\_type = 3;$
    else $join\_type = 2 - lcap;$
    if $((join\_type \equiv 0) \vee (join\_type \equiv 3))$ {
      ⟨Set the incoming and outgoing directions at q; in case of degeneracy set $join\_type$: $= 2$ 597⟩;
      if $(join\_type \equiv 0)$ {
        ⟨If $miterlim$ is less than the secant of half the angle at q then set $join\_type$: $= 2$ 583⟩;
      }
    }
  }

This code is used in section 580.

583.   ⟨If *miterlim* is less than the secant of half the angle at *q* then set *join_type*: = 2 583⟩ ≡
  {
    mp_number *r1*, *r2*;

    new_fraction(*r1*);
    new_fraction(*r2*);
    take_fraction(*r1*, *dxin*, *dxout*);
    take_fraction(*r2*, *dyin*, *dyout*);
    number_add(*r1*, *r2*);
    number_half(*r1*);
    number_add(*r1*, *fraction_half_t*);
    take_fraction(*tmp*, *miterlim*, *r1*);
    if (*number_less*(*tmp*, *unity_t*)) {
      mp_number *ret*;

      new_number(*ret*);
      take_scaled(*ret*, *miterlim*, *tmp*);
      if (*number_less*(*ret*, *unity_t*)) *join_type* = 2;
      free_number(*ret*);
    }
    free_number(*r1*);
    free_number(*r2*);
  }
This code is used in section 582.

584.   ⟨Other local variables for *make_envelope* 584⟩ ≡
  mp_number *tmp*;   /∗ a temporary value ∗/
See also section 592.

This code is used in section 580.

585.   The coordinates of *p* have already been shifted unless *p* is the first knot in which case they get shifted
at the very end.

⟨Add offset *w* to the cubic from *p* to *q* 585⟩ ≡
  number_add(*p*→*right_x*, *w*→*x_coord*);
  number_add(*p*→*right_y*, *w*→*y_coord*);
  number_add(*q*→*left_x*, *w*→*x_coord*);
  number_add(*q*→*left_y*, *w*→*y_coord*);
  number_add(*q*→*x_coord*, *w*→*x_coord*);
  number_add(*q*→*y_coord*, *w*→*y_coord*);
  mp_left_type(*q*) = *mp_explicit*; *mp_right_type*(*q*) = *mp_explicit*
This code is used in section 580.

586.   ⟨Step *w* and move *k* one step closer to *zero_off* 586⟩ ≡
  if (*k* > *zero_off*) {
    w = *mp_next_knot*(*w*);
    decr(*k*);
  }
  else {
    w = *mp_prev_knot*(*w*);
    incr(*k*);
  }
This code is used in section 580.

587.   The cubic from q to the new knot at (x, y) becomes a line segment and the mp_right_x and mp_right_y
fields of r are set from q. This is done in case the cubic containing these control points is "yet to be examined."

$\langle$ Declarations $8\,\rangle +\equiv$
   static mp_knot $mp\_insert\_knot(\textbf{MP } mp, \textbf{mp\_knot } q, \textbf{mp\_number } x, \textbf{mp\_number } y);$

588.   **mp_knot** $mp\_insert\_knot(\textbf{MP } mp, \textbf{mp\_knot } q, \textbf{mp\_number } x, \textbf{mp\_number } y)$
   {   /∗ returns the inserted knot ∗/
      mp_knot r;   /∗ the new knot ∗/
      $r = mp\_new\_knot(mp);$
      $mp\_next\_knot(r) = mp\_next\_knot(q);$
      $mp\_next\_knot(q) = r;$
      $number\_clone(r{\rightarrow}right\_x, q{\rightarrow}right\_x);$
      $number\_clone(r{\rightarrow}right\_y, q{\rightarrow}right\_y);$
      $number\_clone(r{\rightarrow}x\_coord, x);$
      $number\_clone(r{\rightarrow}y\_coord, y);$
      $number\_clone(q{\rightarrow}right\_x, q{\rightarrow}x\_coord);$
      $number\_clone(q{\rightarrow}right\_y, q{\rightarrow}y\_coord);$
      $number\_clone(r{\rightarrow}left\_x, r{\rightarrow}x\_coord);$
      $number\_clone(r{\rightarrow}left\_y, r{\rightarrow}y\_coord);$
      $mp\_left\_type(r) = mp\_explicit;$
      $mp\_right\_type(r) = mp\_explicit;$
      $mp\_originator(r) = mp\_program\_code;$
      return r;
   }

589.   After setting $p{:} = mp\_link(p)$, either $join\_type = 1$ or $q = mp\_link(p).$

$\langle$ Set $p = mp\_link(p)$ and add knots between p and q as required by $join\_type$ 589 $\rangle \equiv$
   {
      $p = mp\_next\_knot(p);$
      if $((join\_type \equiv 0) \vee (join\_type \equiv 3))$ {
         if $(join\_type \equiv 0)$ {$\langle$ Insert a new knot r between p and q as required for a mitered join 590 $\rangle$}
         else {
            $\langle$ Make r the last of two knots inserted between p and q to form a squared join 591 $\rangle;$
         }
         if $(r \neq \Lambda)$ {
            $number\_clone(r{\rightarrow}right\_x, r{\rightarrow}x\_coord);$
            $number\_clone(r{\rightarrow}right\_y, r{\rightarrow}y\_coord);$
         }
      }
   }
This code is used in section 580.

590.   For very small angles, adding a knot is unnecessary and would cause numerical problems, so we just set $r := \Lambda$ in that case.

#define *near_zero_angle_k*   ((**math_data** ∗) *mp→math*)*→near_zero_angle_t*

⟨ Insert a new knot r between p and q as required for a mitered join 590 ⟩ ≡
  {
    mp_number *det*;   /∗ a determinant used for mitered join calculations ∗/
    mp_number *absdet*;
    mp_number *r1*, *r2*;

    new_fraction(*r1*);
    new_fraction(*r2*);
    new_fraction(*det*);
    new_fraction(*absdet*);
    take_fraction(*r1*, *dyout*, *dxin*);
    take_fraction(*r2*, *dxout*, *dyin*);
    set_number_from_substraction(*det*, *r1*, *r2*);
    number_clone(*absdet*, *det*);
    number_abs(*absdet*);
    if (*number_less*(*absdet*, *near_zero_angle_k*)) {
      $r = \Lambda$;   /∗ sine $< 10^{-4}$ ∗/
    }
    else {
      mp_number *xtot*, *ytot*, *xsub*, *ysub*;

      new_fraction(*xsub*);
      new_fraction(*ysub*);
      new_number(*xtot*);
      new_number(*ytot*);
      set_number_from_substraction(*tmp*, *q→x_coord*, *p→x_coord*);
      take_fraction(*r1*, *tmp*, *dyout*);
      set_number_from_substraction(*tmp*, *q→y_coord*, *p→y_coord*);
      take_fraction(*r2*, *tmp*, *dxout*);
      set_number_from_substraction(*tmp*, *r1*, *r2*);
      make_fraction(*r1*, *tmp*, *det*);
      number_clone(*tmp*, *r1*);
      take_fraction(*xsub*, *tmp*, *dxin*);
      take_fraction(*ysub*, *tmp*, *dyin*);
      set_number_from_addition(*xtot*, *p→x_coord*, *xsub*);
      set_number_from_addition(*ytot*, *p→y_coord*, *ysub*);
      $r = mp\_insert\_knot(mp, p, xtot, ytot)$;
      free_number(*xtot*);
      free_number(*ytot*);
      free_number(*xsub*);
      free_number(*ysub*);
    }
    free_number(*r1*);
    free_number(*r2*);
    free_number(*det*);
    free_number(*absdet*);
  }

This code is used in section 589.

591.   ⟨Make r the last of two knots inserted between p and q to form a squared join 591⟩ ≡

{
    mp_number ht_x, ht_y;   /∗ perpendicular to the segment from p to q ∗/
    mp_number ht_x_abs, ht_y_abs;   /∗ absolutes ∗/
    mp_number $xtot$, $ytot$, $xsub$, $ysub$;

    $new\_fraction(xsub)$;
    $new\_fraction(ysub)$;
    $new\_number(xtot)$;
    $new\_number(ytot)$;
    $new\_fraction(ht\_x)$;
    $new\_fraction(ht\_y)$;
    $new\_fraction(ht\_x\_abs)$;
    $new\_fraction(ht\_y\_abs)$;
    $set\_number\_from\_substraction(ht\_x, w{\rightarrow}y\_coord, w0{\rightarrow}y\_coord)$;
    $set\_number\_from\_substraction(ht\_y, w0{\rightarrow}x\_coord, w{\rightarrow}x\_coord)$;
    $number\_clone(ht\_x\_abs, ht\_x)$;
    $number\_clone(ht\_y\_abs, ht\_y)$;
    $number\_abs(ht\_x\_abs)$;
    $number\_abs(ht\_y\_abs)$;
    while $(number\_less(ht\_x\_abs, fraction\_half\_t) \wedge number\_less(ht\_y\_abs, fraction\_half\_t))$ {
        $number\_double(ht\_x)$;
        $number\_double(ht\_y)$;
        $number\_clone(ht\_x\_abs, ht\_x)$;
        $number\_clone(ht\_y\_abs, ht\_y)$;
        $number\_abs(ht\_x\_abs)$;
        $number\_abs(ht\_y\_abs)$;
    }
    ⟨Scan the pen polygon between $w0$ and w and make max_ht the range dot product with
        (ht_x, ht_y) 593⟩;
    {
        mp_number $r1$, $r2$;

        $new\_fraction(r1)$;
        $new\_fraction(r2)$;
        $take\_fraction(r1, dxin, ht\_x)$;
        $take\_fraction(r2, dyin, ht\_y)$;
        $number\_add(r1, r2)$;
        $make\_fraction(tmp, max\_ht, r1)$;
        $free\_number(r1)$;
        $free\_number(r2)$;
    }
    $take\_fraction(xsub, tmp, dxin)$;
    $take\_fraction(ysub, tmp, dyin)$;
    $set\_number\_from\_addition(xtot, p{\rightarrow}x\_coord, xsub)$;
    $set\_number\_from\_addition(ytot, p{\rightarrow}y\_coord, ysub)$;
    $r = mp\_insert\_knot(mp, p, xtot, ytot)$;   /∗ clang: value never read ∗/
    $assert(r)$;
    {
        mp_number $r1$, $r2$;

        $new\_fraction(r1)$;
        $new\_fraction(r2)$;
        $take\_fraction(r1, dxout, ht\_x)$;

$take\_fraction(r2, dyout, ht\_y)$;
$number\_add(r1, r2)$;
$make\_fraction(tmp, max\_ht, r1)$;
$free\_number(r1)$;
$free\_number(r2)$;
}
$take\_fraction(xsub, tmp, dxout)$;
$take\_fraction(ysub, tmp, dyout)$;
$set\_number\_from\_addition(xtot, q \rightarrow x\_coord, xsub)$;
$set\_number\_from\_addition(ytot, q \rightarrow y\_coord, ysub)$;
$r = mp\_insert\_knot(mp, p, xtot, ytot)$;
$free\_number(xsub)$;
$free\_number(ysub)$;
$free\_number(xtot)$;
$free\_number(ytot)$;
$free\_number(ht\_x)$;
$free\_number(ht\_y)$;
$free\_number(ht\_x\_abs)$;
$free\_number(ht\_y\_abs)$;
}

This code is used in section 589.

592. ⟨Other local variables for *make_envelope* 584⟩ +≡
  mp_number *max_ht*;   /∗ maximum height of the pen polygon above the *w0*-*w* line ∗/
  halfword *kk*;   /∗ keeps track of the pen vertices being scanned ∗/
  mp_knot *ww*;   /∗ the pen vertex being tested ∗/

593. The dot product of the vector from *w0* to *ww* with (ht_x, ht_y) ranges from zero to *max_ht*.

⟨Scan the pen polygon between *w0* and *w* and make *max_ht* the range dot product with (ht_x, ht_y) 593⟩ ≡
  $set\_number\_to\_zero(max\_ht)$;
  $kk = zero\_off$;
  $ww = w$;
  while (1) {
    ⟨Step *ww* and move *kk* one step closer to *k0* 594⟩;
    if $(kk \equiv k0)$ **break**;
    {
      mp_number *r1*, *r2*;

      $new\_fraction(r1)$;
      $new\_fraction(r2)$;
      $set\_number\_from\_substraction(tmp, ww \rightarrow x\_coord, w0 \rightarrow x\_coord)$;
      $take\_fraction(r1, tmp, ht\_x)$;
      $set\_number\_from\_substraction(tmp, ww \rightarrow y\_coord, w0 \rightarrow y\_coord)$;
      $take\_fraction(r2, tmp, ht\_y)$;
      $set\_number\_from\_addition(tmp, r1, r2)$;
      $free\_number(r1)$;
      $free\_number(r2)$;
    }
    if $(number\_greater(tmp, max\_ht))$ $number\_clone(max\_ht, tmp)$;
  }

This code is used in section 591.

594.   ⟨Step *ww* and move *kk* one step closer to *k0* 594⟩ ≡
  if (*kk* > *k0*) {
    ww = *mp_next_knot*(*ww*);
    decr(*kk*);
  }
  else {
    ww = *mp_prev_knot*(*ww*);
    incr(*kk*);
  }

This code is used in section 593.

595.   ⟨If endpoint, double the path *c*, and set *spec_p1* and *spec_p2* 595⟩ ≡
  if (*mp_left_type*(*c*) ≡ *mp_endpoint*) {
    mp→*spec_p1* = *mp_htap_ypoc*(*mp*, *c*);
    mp→*spec_p2* = *mp*→*path_tail*;
    mp_originator(*mp*→*spec_p1*) = *mp_program_code*;
    mp_next_knot(*mp*→*spec_p2*) = *mp_next_knot*(*mp*→*spec_p1*);
    mp_next_knot(*mp*→*spec_p1*) = *c*;
    mp_remove_cubic(*mp*, *mp*→*spec_p1*);
    c = *mp*→*spec_p1*;
    if (*c* ≠ *mp_next_knot*(*c*)) {
      mp_originator(*mp*→*spec_p2*) = *mp_program_code*;
      mp_remove_cubic(*mp*, *mp*→*spec_p2*);
    }
    else {
      ⟨Make *c* look like a cycle of length one 596⟩;
    }
  }

This code is used in section 580.

596.   ⟨Make *c* look like a cycle of length one 596⟩ ≡
  {
    mp_left_type(*c*) = *mp_explicit*;
    mp_right_type(*c*) = *mp_explicit*;
    number_clone(*c*→*left_x*, *c*→*x_coord*);
    number_clone(*c*→*left_y*, *c*→*y_coord*);
    number_clone(*c*→*right_x*, *c*→*x_coord*);
    number_clone(*c*→*right_y*, *c*→*y_coord*);
  }

This code is used in section 595.

597.   In degenerate situations we might have to look at the knot preceding q. That knot is p but if $p <> c$, its coordinates have already been offset by w.

$\langle$ Set the incoming and outgoing directions at q; in case of degeneracy set $\mathit{join\_type}\colon = 2$ 597 $\rangle \equiv$

```
{
    set_number_from_substraction(dxin, q→x_coord, q→left_x);
    set_number_from_substraction(dyin, q→y_coord, q→left_y);
    if (number_zero(dxin) ∧ number_zero(dyin)) {
        set_number_from_substraction(dxin, q→x_coord, p→right_x);
        set_number_from_substraction(dyin, q→y_coord, p→right_y);
        if (number_zero(dxin) ∧ number_zero(dyin)) {
            set_number_from_substraction(dxin, q→x_coord, p→x_coord);
            set_number_from_substraction(dyin, q→y_coord, p→y_coord);
            if (p ≠ c) {      /∗ the coordinates of p have been offset by w ∗/
                number_add(dxin, w→x_coord);
                number_add(dyin, w→y_coord);
            }
        }
    }
    pyth_add(tmp, dxin, dyin);
    if (number_zero(tmp)) {
        join_type = 2;
    }
    else {
        mp_number r1;

        new_fraction(r1);
        make_fraction(r1, dxin, tmp);
        number_clone(dxin, r1);
        make_fraction(r1, dyin, tmp);
        number_clone(dyin, r1);
        free_number(r1);
        ⟨ Set the outgoing direction at q 598 ⟩;
    }
}
```

This code is used in section 582.

598.   If $q = c$ then the coordinates of r and the control points between q and r have already been offset by h.

$\langle$ Set the outgoing direction at q 598 $\rangle \equiv$

```
{
  set_number_from_substraction(dxout, q→right_x, q→x_coord);
  set_number_from_substraction(dyout, q→right_y, q→y_coord);
  if (number_zero(dxout) ∧ number_zero(dyout)) {
    r = mp_next_knot(q);
    set_number_from_substraction(dxout, r→left_x, q→x_coord);
    set_number_from_substraction(dyout, r→left_y, q→y_coord);
    if (number_zero(dxout) ∧ number_zero(dyout)) {
      set_number_from_substraction(dxout, r→x_coord, q→x_coord);
      set_number_from_substraction(dyout, r→y_coord, q→y_coord);
    }
  }
  if (q ≡ c) {
    number_substract(dxout, h→x_coord);
    number_substract(dyout, h→y_coord);
  }
  pyth_add(tmp, dxout, dyout);
  if (number_zero(tmp)) {      /* mp_confusion(mp, "degenerate␣spec"); */
    ;      /* But apparently, it actually can happen. The test case is this: path p; linejoin := mitered;
              p:= (10,0)..(0,10)..(-10,0)..(0,-10)..cycle; addto currentpicture contour p withpen pensquare;
              The reason for failure here is the addition of r ≠ q in revision 1757 in "Advance p to node q,
              removing any "dead" cubics", which itself was needed to fix a bug with disappearing knots in a
              path that was rotated exactly 45 degrees (luatex.org bug 530).  */
  }
  else {
    mp_number r1;

    new_fraction(r1);
    make_fraction(r1, dxout, tmp);
    number_clone(dxout, r1);
    make_fraction(r1, dyout, tmp);
    number_clone(dyout, r1);
    free_number(r1);
  }
}
```

This code is used in section 597.

599.   Direction and intersection times.   A path of length n is defined parametrically by functions $x(t)$ and $y(t)$, for $0 \leq t \leq n$; we can regard t as the "time" at which the path reaches the point $\big(x(t), y(t)\big)$. In this section of the program we shall consider operations that determine special times associated with given paths: the first time that a path travels in a given direction, and a pair of times at which two paths cross each other.

600.   Let's start with the easier task. The function *find_direction_time* is given a direction (x, y) and a path starting at h. If the path never travels in direction (x, y), the direction time will be -1; otherwise it will be nonnegative.

Certain anomalous cases can arise: If $(x, y) = (0, 0)$, so that the given direction is undefined, the direction time will be 0. If $(x'(t), y'(t)) = (0, 0)$, so that the path direction is undefined, it will be assumed to match any given direction at time t.

The routine solves this problem in nondegenerate cases by rotating the path and the given direction so that $(x, y) = (1, 0)$; i.e., the main task will be to find when a given path first travels "due east."

static void *mp_find_direction_time*(**MP** *mp*, **mp_number** *∗ret*, **mp_number** *x_orig*, **mp_number**
        y_orig, **mp_knot** *h*)
{
  mp_number *max*;   /∗ max$(|x|, |y|)$ ∗/
  mp_knot *p*, *q*;   /∗ for list traversal ∗/
  mp_number *n*;   /∗ the direction time at knot *p* ∗/
  mp_number *tt*;   /∗ the direction time within a cubic ∗/
  mp_number *x*, *y*;
  mp_number *abs_x*, *abs_y*;   /∗ Other local variables for *find_direction_time* ∗/
  mp_number *x1*, *x2*, *x3*, *y1*, *y2*, *y3*;   /∗ multiples of rotated derivatives ∗/
  mp_number *phi*;   /∗ angles of exit and entry at a knot ∗/
  mp_number *t*;   /∗ temp storage ∗/
  mp_number *ab_vs_cd*;
  new_number(*max*);
  new_number(*x1*);
  new_number(*x2*);
  new_number(*x3*);
  new_number(*y1*);
  new_number(*y2*);
  new_number(*y3*);
  new_fraction(*t*);
  new_angle(*phi*);
  new_number(*ab_vs_cd*);
  set_number_to_zero(*∗ret*);   /∗ just in case ∗/
  new_number(*x*);
  new_number(*y*);
  new_number(*abs_x*);
  new_number(*abs_y*);
  new_number(*n*);
  new_fraction(*tt*);
  number_clone(*x*, *x_orig*);
  number_clone(*y*, *y_orig*);
  number_clone(*abs_x*, *x_orig*);
  number_clone(*abs_y*, *y_orig*);
  number_abs(*abs_x*);
  number_abs(*abs_y*);
    /∗ Normalize the given direction for better accuracy; but **return** with zero result if it's zero ∗/
  if (*number_less*(*abs_x*, *abs_y*)) {
  mp_number *r1*;
  new_fraction(*r1*);
  make_fraction(*r1*, *x*, *abs_y*);
  number_clone(*x*, *r1*);
  free_number(*r1*);

```
    if (number_positive(y)) {
        number_clone(y, fraction_one_t);
    }
    else {
        number_clone(y, fraction_one_t);
        number_negate(y);
    }
}
else if (number_zero(x)) {
    goto FREE;
}
else {
    mp_number r1;

    new_fraction(r1);
    make_fraction(r1, y, abs_x);
    number_clone(y, r1);
    free_number(r1);
    if (number_positive(x)) {
        number_clone(x, fraction_one_t);
    }
    else {
        number_clone(x, fraction_one_t);
        number_negate(x);
    }
}
p = h;
while (1) {
    if (mp_right_type(p) ≡ mp_endpoint) break;
    q = mp_next_knot(p);
    ⟨Rotate the cubic between p and q; then goto found if the rotated cubic travels due east at some
         time tt; but break if an entire cyclic path has been traversed 601⟩;
    p = q;
    number_add(n, unity_t);
}
set_number_to_unity(*ret);
number_negate(*ret);
goto FREE;
FOUND: set_number_from_addition(*ret, n, tt);
goto FREE;
FREE: free_number(x);
free_number(y);
free_number(abs_x);
free_number(abs_y);         /∗ Free local variables for find_direction_time ∗/
free_number(x1);
free_number(x2);
free_number(x3);
free_number(y1);
free_number(y2);
free_number(y3);
free_number(t);
free_number(phi);
free_number(ab_vs_cd);
```

    $\mathit{free\_number}\,(n)$;
    $\mathit{free\_number}\,(\mathit{max})$;
    $\mathit{free\_number}\,(\mathit{tt})$;
  }

601.   Since we're interested in the tangent directions, we work with the derivative

$$\frac{1}{3}B'(x_0, x_1, x_2, x_3; t) = B(x_1 - x_0, x_2 - x_1, x_3 - x_2; t)$$

instead of $B(x_0, x_1, x_2, x_3; t)$ itself. The derived coefficients are also scale-d up in order to achieve better accuracy.

The given path may turn abruptly at a knot, and it might pass the critical tangent direction at such a time. Therefore we remember the direction *phi* in which the previous rotated cubic was traveling. (The value of *phi* will be undefined on the first cubic, i.e., when $n = 0$.)

#define *we_found_it*
                {
                    number_clone(*tt*, *t*);
                    fraction_to_round_scaled(*tt*);
                    goto FOUND;
                }

⟨ Rotate the cubic between *p* and *q*; then **goto** *found* if the rotated cubic travels due east at some time *tt*;
        but **break** if an entire cyclic path has been traversed 601 ⟩ ≡
    set_number_to_zero(*tt*);   /∗ Set local variables *x1*, *x2*, *x3* and *y1*, *y2*, *y3* to multiples of the control
        points of the rotated derivatives ∗/
    {
        mp_number *absval*;

        new_number(*absval*);
        set_number_from_substraction(*x1*, *p⃗right_x*, *p⃗x_coord*);
        set_number_from_substraction(*x2*, *q⃗left_x*, *p⃗right_x*);
        set_number_from_substraction(*x3*, *q⃗x_coord*, *q⃗left_x*);
        set_number_from_substraction(*y1*, *p⃗right_y*, *p⃗y_coord*);
        set_number_from_substraction(*y2*, *q⃗left_y*, *p⃗right_y*);
        set_number_from_substraction(*y3*, *q⃗y_coord*, *q⃗left_y*);
        number_clone(*absval*, *x2*);
        number_abs(*absval*);
        number_clone(*max*, *x1*);
        number_abs(*max*);
        if (*number_greater*(*absval*, *max*)) {
            number_clone(*max*, *absval*);
        }
        number_clone(*absval*, *x3*);
        number_abs(*absval*);
        if (*number_greater*(*absval*, *max*)) {
            number_clone(*max*, *absval*);
        }
        number_clone(*absval*, *y1*);
        number_abs(*absval*);
        if (*number_greater*(*absval*, *max*)) {
            number_clone(*max*, *absval*);
        }
        number_clone(*absval*, *y2*);
        number_abs(*absval*);
        if (*number_greater*(*absval*, *max*)) {
            number_clone(*max*, *absval*);
        }
        number_clone(*absval*, *y3*);

    $number\_abs(absval)$;
    if ($number\_greater(absval, max)$) {
        $number\_clone(max, absval)$;
    }
    $free\_number(absval)$;
    if ($number\_zero(max)$) **goto** FOUND;
    while ($number\_less(max, fraction\_half\_t)$) {
        $number\_double(max)$;
        $number\_double(x1)$;
        $number\_double(x2)$;
        $number\_double(x3)$;
        $number\_double(y1)$;
        $number\_double(y2)$;
        $number\_double(y3)$;
    }
    $number\_clone(t, x1)$;
    {
        mp_number $r1$, $r2$;

        $new\_fraction(r1)$;
        $new\_fraction(r2)$;
        $take\_fraction(r1, x1, x)$;
        $take\_fraction(r2, y1, y)$;
        $set\_number\_from\_addition(x1, r1, r2)$;
        $take\_fraction(r1, y1, x)$;
        $take\_fraction(r2, t, y)$;
        $set\_number\_from\_substraction(y1, r1, r2)$;
        $number\_clone(t, x2)$;
        $take\_fraction(r1, x2, x)$;
        $take\_fraction(r2, y2, y)$;
        $set\_number\_from\_addition(x2, r1, r2)$;
        $take\_fraction(r1, y2, x)$;
        $take\_fraction(r2, t, y)$;
        $set\_number\_from\_substraction(y2, r1, r2)$;
        $number\_clone(t, x3)$;
        $take\_fraction(r1, x3, x)$;
        $take\_fraction(r2, y3, y)$;
        $set\_number\_from\_addition(x3, r1, r2)$;
        $take\_fraction(r1, y3, x)$;
        $take\_fraction(r2, t, y)$;
        $set\_number\_from\_substraction(y3, r1, r2)$;
        $free\_number(r1)$;
        $free\_number(r2)$;
    }
}
if ($number\_zero(y1)$)
    if ($number\_zero(x1) \vee number\_positive(x1)$) **goto** FOUND;
if ($number\_positive(n)$) {   /∗ Exit to *found* if an eastward direction occurs at knot p ∗/
    mp_number $theta$;
    mp_number tmp;

    $new\_angle(theta)$;
    $n\_arg(theta, x1, y1)$;
    $new\_angle(tmp)$;

$set\_number\_from\_substraction(tmp, theta, one\_eighty\_deg\_t);$
if $(number\_nonnegative(theta) \wedge number\_nonpositive(phi) \wedge number\_greaterequal(phi, tmp))$ {
  $free\_number(tmp);$
  $free\_number(theta);$
  goto FOUND;
}
$set\_number\_from\_addition(tmp, theta, one\_eighty\_deg\_t);$
if $(number\_nonpositive(theta) \wedge number\_nonnegative(phi) \wedge number\_lessequal(phi, tmp))$ {
  $free\_number(tmp);$
  $free\_number(theta);$
  goto FOUND;
}
$free\_number(tmp);$
$free\_number(theta);$
if $(p \equiv h)$ **break**;
}
if $(number\_nonzero(x3) \vee number\_nonzero(y3))$ {
  $n\_arg(phi, x3, y3);$
}   /∗ Exit to *found* if the curve whose derivatives are specified by *x1*, *x2*, *x3*, *y1*, *y2*, *y3* travels
      eastward at some time *tt* ∗/   /∗ In this step we want to use the *crossing_point* routine to
      find the roots of the quadratic equation $B(y_1, y_2, y_3; t) = 0$. Several complications arise: If the
      quadratic equation has a double root, the curve never crosses zero, and *crossing_point* will find
      nothing; this case occurs iff $y_1 y_3 = y_2^2$ and $y_1 y_2 < 0$. If the quadratic equation has simple roots, or
      only one root, we may have to negate it so that $B(y_1, y_2, y_3; t)$ crosses from positive to negative at
      its first root. And finally, we need to do special things if $B(y_1, y_2, y_3; t)$ is identically zero. ∗/
if $(number\_negative(x1))$
  if $(number\_negative(x2))$
    if $(number\_negative(x3))$ **goto** DONE;
{
  $ab\_vs\_cd(ab\_vs\_cd, y1, y3, y2, y2);$
  if $(number\_zero(ab\_vs\_cd))$ {
    /∗ Handle the test for eastward directions when $y_1 y_3 = y_2^2$; either **goto** *found* or **goto** *done* ∗/
    {
      $ab\_vs\_cd(ab\_vs\_cd, y1, y2, zero\_t, zero\_t);$
      if $(number\_negative(ab\_vs\_cd))$ {
        mp_number $tmp, arg2;$

        $new\_number(tmp);$
        $new\_number(arg2);$
        $set\_number\_from\_substraction(arg2, y1, y2);$
        $make\_fraction(t, y1, arg2);$
        $free\_number(arg2);$
        $set\_number\_from\_of\_the\_way(x1, t, x1, x2);$
        $set\_number\_from\_of\_the\_way(x2, t, x2, x3);$
        $set\_number\_from\_of\_the\_way(tmp, t, x1, x2);$
        if $(number\_zero(tmp) \vee number\_positive(tmp))$ {
          $free\_number(tmp);$
          $we\_found\_it;$
        }
        $free\_number(tmp);$
      }
      else if $(number\_zero(y3))$ {

     if (*number_zero*(*y1*)) {   /∗ Exit to *found* if the derivative $B(x_1, x_2, x_3; t)$ becomes ≥ 0 ∗/
           /∗ At this point we know that the derivative of $y(t)$ is identically zero, and that *x1* < 0;
              but either *x2* ≥ 0 or *x3* ≥ 0, so there's some hope of traveling east. ∗/
       {

         mp_number *arg1*, *arg2*, *arg3*;

         new_number(*arg1*);
         new_number(*arg2*);
         new_number(*arg3*);
         number_clone(*arg1*, *x1*);
         number_negate(*arg1*);
         number_clone(*arg2*, *x2*);
         number_negate(*arg2*);
         number_clone(*arg3*, *x3*);
         number_negate(*arg3*);
         crossing_point(*t*, *arg1*, *arg2*, *arg3*);
         free_number(*arg1*);
         free_number(*arg2*);
         free_number(*arg3*);
         if (*number_lessequal*(*t*, *fraction_one_t*)) *we_found_it*;
         ab_vs_cd(*ab_vs_cd*, *x1*, *x3*, *x2*, *x2*);
         if (*number_nonpositive*(*ab_vs_cd*)) {
           mp_number *arg2*;

           new_number(*arg2*);
           set_number_from_substraction(*arg2*, *x1*, *x2*);
           make_fraction(*t*, *x1*, *arg2*);
           free_number(*arg2*);
           we_found_it;
         }
       }
      }
     else if (*number_zero*(*x3*) ∨ *number_positive*(*x3*)) {
      set_number_to_unity(*tt*);
      goto FOUND;
     }
    }
   goto DONE;
  }
 }
}
if (*number_zero*(*y1*) ∨ *number_negative*(*y1*)) {
  if (*number_negative*(*y1*)) {
   number_negate(*y1*);
   number_negate(*y2*);
   number_negate(*y3*);
  }
  else if (*number_positive*(*y2*)) {
   number_negate(*y2*);
   number_negate(*y3*);
  }
}   /∗ Check the places where $B(y_1, y_2, y_3; t) = 0$ to see if $B(x_1, x_2, x_3; t) \geq 0$ ∗/
    /∗ The quadratic polynomial $B(y_1, y_2, y_3; t)$ begins ≥ 0 and has at most two roots, because we know
      that it isn't identically zero. It must be admitted that the *crossing_point* routine is not perfectly

accurate; rounding errors might cause it to find a root when $y_1 y_3 > y_2^2$, or to miss the roots when $y_1 y_3 < y_2^2$. The rotation process is itself subject to rounding errors. Yet this code optimistically tries to do the right thing. $*/$

$crossing\_point(t, y1, y2, y3)$;
if $(number\_greater(t, fraction\_one\_t))$ **goto** DONE;
$set\_number\_from\_of\_the\_way(y2, t, y2, y3)$;
$set\_number\_from\_of\_the\_way(x1, t, x1, x2)$;
$set\_number\_from\_of\_the\_way(x2, t, x2, x3)$;
$set\_number\_from\_of\_the\_way(x1, t, x1, x2)$;
if $(number\_zero(x1) \lor number\_positive(x1))$ we_found_it;
if $(number\_positive(y2))$ $set\_number\_to\_zero(y2)$;
$number\_clone(tt, t)$;
{
    mp_number $arg1$, $arg2$, $arg3$;

    $new\_number(arg1)$;
    $new\_number(arg2)$;
    $new\_number(arg3)$;
    $number\_clone(arg2, y2)$;
    $number\_negate(arg2)$;
    $number\_clone(arg3, y3)$;
    $number\_negate(arg3)$;
    $crossing\_point(t, arg1, arg2, arg3)$;
    $free\_number(arg1)$;
    $free\_number(arg2)$;
    $free\_number(arg3)$;
}
if $(number\_greater(t, fraction\_one\_t))$ **goto** DONE;
{
    mp_number tmp;

    $new\_number(tmp)$;
    $set\_number\_from\_of\_the\_way(x1, t, x1, x2)$;
    $set\_number\_from\_of\_the\_way(x2, t, x2, x3)$;
    $set\_number\_from\_of\_the\_way(tmp, t, x1, x2)$;
    if $(number\_nonnegative(tmp))$ {
        $free\_number(tmp)$;
        $set\_number\_from\_of\_the\_way(t, t, tt, fraction\_one\_t)$;
        we_found_it;
    }
    $free\_number(tmp)$;
}
DONE:

This code is used in section 600.

602.   The intersection of two cubics can be found by an interesting variant of the general bisection scheme described in the introduction to *crossing_point*. Given $w(t) = B(w_0, w_1, w_2, w_3; t)$ and $z(t) = B(z_0, z_1, z_2, z_3; t)$, we wish to find a pair of times (t_1, t_2) such that $w(t_1) = z(t_2)$, if an intersection exists. First we find the smallest rectangle that encloses the points $\{w_0, w_1, w_2, w_3\}$ and check that it overlaps the smallest rectangle that encloses $\{z_0, z_1, z_2, z_3\}$; if not, the cubics certainly don't intersect. But if the rectangles do overlap, we bisect the intervals, getting new cubics w' and w'', z' and z''; the intersection routine first tries for an intersection between w' and z', then (if unsuccessful) between w' and z'', then (if still unsuccessful) between w'' and z', finally (if thrice unsuccessful) between w'' and z''. After l successful levels of bisection we will have determined the intersection times t_1 and t_2 to l bits of accuracy.

As before, it is better to work with the numbers $W_k = 2^l(w_k - w_{k-1})$ and $Z_k = 2^l(z_k - z_{k-1})$ rather than the coefficients w_k and z_k themselves. We also need one other quantity, $\Delta = 2^l(w_0 - z_0)$, to determine when the enclosing rectangles overlap. Here's why: The x coordinates of $w(t)$ are between $u_{\min}$ and $u_{\max}$, and the x coordinates of $z(t)$ are between $x_{\min}$ and $x_{\max}$, if we write $w_k = (u_k, v_k)$ and $z_k = (x_k, y_k)$ and $u_{\min} = \min(u_0, u_1, u_2, u_3)$, etc. These intervals of x coordinates overlap if and only if $u_{\min} \mathbin{\mathrm{L}} x_{\max}$ and $x_{\min} \mathbin{\mathrm{L}} u_{\max}$. Letting

$$U_{\min} = \min(0, U_1, U_1 + U_2, U_1 + U_2 + U_3), \ U_{\max} = \max(0, U_1, U_1 + U_2, U_1 + U_2 + U_3),$$

we have $2^l u_{\min} = 2^l u_0 + U_{\min}$, etc.; the condition for overlap reduces to

$$X_{\min} - U_{\max} \mathbin{\mathrm{L}} 2^l(u_0 - x_0) \mathbin{\mathrm{L}} X_{\max} - U_{\min}.$$

Thus we want to maintain the quantity $2^l(u_0 - x_0)$; similarly, the quantity $2^l(v_0 - y_0)$ accounts for the y coordinates. The coordinates of $\Delta = 2^l(w_0 - z_0)$ must stay bounded as l increases, because of the overlap condition; i.e., we know that $X_{\min}$, $X_{\max}$, and their relatives are bounded, hence $X_{\max} - U_{\min}$ and $X_{\min} - U_{\max}$ are bounded.

603.   Incidentally, if the given cubics intersect more than once, the process just sketched will not necessarily find the lexicographically smallest pair (t_1, t_2). The solution actually obtained will be smallest in "shuffled order"; i.e., if $t_1 = (.a_1 a_2 \ldots a_{16})_2$ and $t_2 = (.b_1 b_2 \ldots b_{16})_2$, then we will minimize $a_1 b_1 a_2 b_2 \ldots a_{16} b_{16}$, not $a_1 a_2 \ldots a_{16} b_1 b_2 \ldots b_{16}$. Shuffled order agrees with lexicographic order if all pairs of solutions (t_1, t_2) and (t_1', t_2') have the property that $t_1 < t_1'$ iff $t_2 < t_2'$; but in general, lexicographic order can be quite different, and the bisection algorithm would be substantially less efficient if it were constrained by lexicographic order.

For example, suppose that an overlap has been found for $l = 3$ and $(t_1, t_2) = (.101, .011)$ in binary, but that no overlap is produced by either of the alternatives $(.1010, .0110)$, $(.1010, .0111)$ at level 4. Then there is probably an intersection in one of the subintervals $(.1011, .011x)$; but lexicographic order would require us to explore $(.1010, .1xxx)$ and $(.1011, .00xx)$ and $(.1011, .010x)$ first. We wouldn't want to store all of the subdivision data for the second path, so the subdivisions would have to be regenerated many times. Such inefficiencies would be associated with every '1' in the binary representation of t_1.

604.   The subdivision process introduces rounding errors, hence we need to make a more liberal test for overlap. It is not hard to show that the computed values of U_i differ from the truth by at most l, on level l, hence $U_{\min}$ and $U_{\max}$ will be at most $3l$ in error. If β is an upper bound on the absolute error in the computed components of $\Delta = (delx, dely)$ on level l, we will replace the test '$X_{\min} - U_{\max} \mathbin{\mathrm{L}} delx$' by the more liberal test '$X_{\min} - U_{\max} \mathbin{\mathrm{L}} delx + tol$', where $tol = 6l + \beta$.

More accuracy is obtained if we try the algorithm first with $tol = 0$; the more liberal tolerance is used only if an exact approach fails. It is convenient to do this double-take by letting '3' in the preceding paragraph be a parameter, which is first 0, then 3.

⟨ Global variables 14 ⟩ +≡
  unsigned int *tol_step*;   /∗ either 0 or 3, usually ∗/

605.   We shall use an explicit stack to implement the recursive bisection method described above. The *bisect_stack* array will contain numerous 5-word packets like $(U_1, U_2, U_3, U_{min}, U_{max})$, as well as 20-word packets comprising the 5-word packets for U, V, X, and Y.

   The following macros define the allocation of stack positions to the quantities needed for bisection-intersection.

#define *stack_1*(A)   *mp→bisect_stack*[(A)]   /∗ U_1, V_1, X_1, or Y_1 ∗/
#define *stack_2*(A)   *mp→bisect_stack*[(A) + 1]   /∗ U_2, V_2, X_2, or Y_2 ∗/
#define *stack_3*(A)   *mp→bisect_stack*[(A) + 2]   /∗ U_3, V_3, X_3, or Y_3 ∗/
#define *stack_min*(A)   *mp→bisect_stack*[(A) + 3]   /∗ U_{min}, V_{min}, X_{min}, or Y_{min} ∗/
#define *stack_max*(A)   *mp→bisect_stack*[(A) + 4]   /∗ U_{max}, V_{max}, X_{max}, or Y_{max} ∗/
#define *int_packets* 20   /∗ number of words to represent U_k, V_k, X_k, and Y_k ∗/

#define *u_packet*(A)   ((A) − 5)
#define *v_packet*(A)   ((A) − 10)
#define *x_packet*(A)   ((A) − 15)
#define *y_packet*(A)   ((A) − 20)
#define *l_packets*   (*mp→bisect_ptr* − *int_packets*)
#define *r_packets*   *mp→bisect_ptr*
#define *ul_packet*   *u_packet*(*l_packets*)   /∗ base of U'_k variables ∗/
#define *vl_packet*   *v_packet*(*l_packets*)   /∗ base of V'_k variables ∗/
#define *xl_packet*   *x_packet*(*l_packets*)   /∗ base of X'_k variables ∗/
#define *yl_packet*   *y_packet*(*l_packets*)   /∗ base of Y'_k variables ∗/
#define *ur_packet*   *u_packet*(*r_packets*)   /∗ base of U''_k variables ∗/
#define *vr_packet*   *v_packet*(*r_packets*)   /∗ base of V''_k variables ∗/
#define *xr_packet*   *x_packet*(*r_packets*)   /∗ base of X''_k variables ∗/
#define *yr_packet*   *y_packet*(*r_packets*)   /∗ base of Y''_k variables ∗/

#define *u1l*   *stack_1*(*ul_packet*)   /∗ U'_1 ∗/
#define *u2l*   *stack_2*(*ul_packet*)   /∗ U'_2 ∗/
#define *u3l*   *stack_3*(*ul_packet*)   /∗ U'_3 ∗/
#define *v1l*   *stack_1*(*vl_packet*)   /∗ V'_1 ∗/
#define *v2l*   *stack_2*(*vl_packet*)   /∗ V'_2 ∗/
#define *v3l*   *stack_3*(*vl_packet*)   /∗ V'_3 ∗/
#define *x1l*   *stack_1*(*xl_packet*)   /∗ X'_1 ∗/
#define *x2l*   *stack_2*(*xl_packet*)   /∗ X'_2 ∗/
#define *x3l*   *stack_3*(*xl_packet*)   /∗ X'_3 ∗/
#define *y1l*   *stack_1*(*yl_packet*)   /∗ Y'_1 ∗/
#define *y2l*   *stack_2*(*yl_packet*)   /∗ Y'_2 ∗/
#define *y3l*   *stack_3*(*yl_packet*)   /∗ Y'_3 ∗/
#define *u1r*   *stack_1*(*ur_packet*)   /∗ U''_1 ∗/
#define *u2r*   *stack_2*(*ur_packet*)   /∗ U''_2 ∗/
#define *u3r*   *stack_3*(*ur_packet*)   /∗ U''_3 ∗/
#define *v1r*   *stack_1*(*vr_packet*)   /∗ V''_1 ∗/
#define *v2r*   *stack_2*(*vr_packet*)   /∗ V''_2 ∗/
#define *v3r*   *stack_3*(*vr_packet*)   /∗ V''_3 ∗/
#define *x1r*   *stack_1*(*xr_packet*)   /∗ X''_1 ∗/
#define *x2r*   *stack_2*(*xr_packet*)   /∗ X''_2 ∗/
#define *x3r*   *stack_3*(*xr_packet*)   /∗ X''_3 ∗/
#define *y1r*   *stack_1*(*yr_packet*)   /∗ Y''_1 ∗/
#define *y2r*   *stack_2*(*yr_packet*)   /∗ Y''_2 ∗/
#define *y3r*   *stack_3*(*yr_packet*)   /∗ Y''_3 ∗/

#define *stack_dx*   *mp→bisect_stack*[*mp→bisect_ptr*]   /∗ stacked value of *delx* ∗/
#define *stack_dy*   *mp→bisect_stack*[*mp→bisect_ptr* + 1]   /∗ stacked value of *dely* ∗/

#define   *stack_tol*   *mp→bisect_stack*[*mp→bisect_ptr* + 2]   /∗ stacked value of *tol* ∗/
#define   *stack_uv*   *mp→bisect_stack*[*mp→bisect_ptr* + 3]   /∗ stacked value of *uv* ∗/
#define   *stack_xy*   *mp→bisect_stack*[*mp→bisect_ptr* + 4]   /∗ stacked value of *xy* ∗/
#define   *int_increment*   (*int_packets* + *int_packets* + 5)   /∗ number of stack words per level ∗/

⟨ Global variables 14 ⟩ +≡
  mp_number ∗*bisect_stack*;
  integer *bisect_ptr*;

606.   ⟨ Allocate or initialize variables 28 ⟩ +≡
  mp→bisect_stack = *xmalloc*((*bistack_size* + 1), **sizeof**(**mp_number**));
  {
    int *i*;
    for (*i* = 0; *i* < *bistack_size* + 1; *i*++) {
      new_number(*mp→bisect_stack*[*i*]);
    }
  }

607.   ⟨ Dealloc variables 27 ⟩ +≡
  {
    int *i*;
    for (*i* = 0; *i* < *bistack_size* + 1; *i*++) {
      free_number(*mp→bisect_stack*[*i*]);
    }
  }
  xfree(*mp→bisect_stack*);

608.   ⟨ Check the "constant" values for consistency 30 ⟩ +≡
  if (*int_packets* + 17 ∗ *int_increment* > *bistack_size*) *mp→bad* = 19;

609.   Computation of the min and max is a tedious but fairly fast sequence of instructions; exactly four comparisons are made in each branch.

#define *set_min_max*(*A*)   *debug_number*(*stack_1*(*A*));
        debug_number(*stack_3*(*A*));
        debug_number(*stack_2*(*A*));
        debug_number(*stack_min*(*A*));
        debug_number(*stack_max*(*A*));
        if (*number_negative*(*stack_1*((*A*)))) {
            if (*number_nonnegative*(*stack_3*((*A*)))) {
                if (*number_negative*(*stack_2*((*A*))))
                    set_number_from_addition(*stack_min*((*A*)), *stack_1*((*A*)), *stack_2*((*A*)));
                else *number_clone*(*stack_min*((*A*)), *stack_1*((*A*)));
                set_number_from_addition(*stack_max*((*A*)), *stack_1*((*A*)), *stack_2*((*A*)));
                number_add(*stack_max*((*A*)), *stack_3*((*A*)));
                if (*number_negative*(*stack_max*((*A*)))) *set_number_to_zero*(*stack_max*((*A*)));
            }
            else {
                set_number_from_addition(*stack_min*((*A*)), *stack_1*((*A*)), *stack_2*((*A*)));
                number_add(*stack_min*((*A*)), *stack_3*((*A*)));
                if (*number_greater*(*stack_min*((*A*)), *stack_1*((*A*))))
                    number_clone(*stack_min*((*A*)), *stack_1*((*A*)));
                set_number_from_addition(*stack_max*((*A*)), *stack_1*((*A*)), *stack_2*((*A*)));
                if (*number_negative*(*stack_max*((*A*)))) *set_number_to_zero*(*stack_max*((*A*)));
            }
        }
        else if (*number_nonpositive*(*stack_3*((*A*)))) {
            if (*number_positive*(*stack_2*((*A*))))
                set_number_from_addition(*stack_max*((*A*)), *stack_1*((*A*)), *stack_2*((*A*)));
            else *number_clone*(*stack_max*((*A*)), *stack_1*((*A*)));
            set_number_from_addition(*stack_min*((*A*)), *stack_1*((*A*)), *stack_2*((*A*)));
            number_add(*stack_min*((*A*)), *stack_3*((*A*)));
            if (*number_positive*(*stack_min*((*A*)))) *set_number_to_zero*(*stack_min*((*A*)));
        }
        else {
            set_number_from_addition(*stack_max*((*A*)), *stack_1*((*A*)), *stack_2*((*A*)));
            number_add(*stack_max*((*A*)), *stack_3*((*A*)));
            if (*number_less*(*stack_max*((*A*)), *stack_1*((*A*)))) *number_clone*(*stack_max*((*A*)), *stack_1*((*A*)));
            set_number_from_addition(*stack_min*((*A*)), *stack_1*((*A*)), *stack_2*((*A*)));
            if (*number_positive*(*stack_min*((*A*)))) *set_number_to_zero*(*stack_min*((*A*)));
        }

610.   It's convenient to keep the current values of l, t_1, and t_2 in the integer form $2^l + 2^l t_1$ and $2^l + 2^l t_2$. The *cubic_intersection* routine uses global variables *cur_t* and *cur_tt* for this purpose; after successful completion, *cur_t* and *cur_tt* will contain *unity* plus the *scaled* values of t_1 and t_2.

The values of *cur_t* and *cur_tt* will be set to zero if *cubic_intersection* finds no intersection. The routine gives up and gives an approximate answer if it has backtracked more than 5000 times (otherwise there are cases where several minutes of fruitless computation would be possible).

#define *max_patience* 5000

⟨ Global variables 14 ⟩ +≡
  mp_number *cur_t*;
  mp_number *cur_tt*;   /∗ controls and results of *cubic_intersection* ∗/
  integer *time_to_go*;   /∗ this many backtracks before giving up ∗/
  mp_number *max_t*;   /∗ maximum of 2^{l+1} so far achieved ∗/

611.   ⟨ Initialize table entries 182 ⟩ +≡
  new_number(*mp*→*cur_t*);
  new_number(*mp*→*cur_tt*);
  new_number(*mp*→*max_t*);

612.   ⟨ Dealloc variables 27 ⟩ +≡
  free_number(*mp*→*cur_t*);
  free_number(*mp*→*cur_tt*);
  free_number(*mp*→*max_t*);

613.   The given cubics $B(w_0, w_1, w_2, w_3; t)$ and $B(z_0, z_1, z_2, z_3; t)$ are specified in adjacent knot nodes $(p, \mathit{mp\_link}(p))$ and $(pp, \mathit{mp\_link}(pp))$, respectively.

#define $\mathit{half}(A)$ $((A)/2)$

```
  static void mp_cubic_intersection(MP mp, mp_knot p, mp_knot pp)
  {
    mp_knot q, qq;      /* mp_link(p), mp_link(pp) */
    mp→time_to_go = max_patience;
    set_number_from_scaled(mp→max_t, 2);
    ⟨Initialize for intersections at level zero 617⟩;
  CONTINUE:
    while (1) {
      if (number_to_scaled(mp→delx) − mp→tol ≤ number_to_scaled(stack_max(x_packet(mp→xy))) −
            number_to_scaled(stack_min(u_packet(mp→uv))))
        if (number_to_scaled(mp→delx) + mp→tol ≥ number_to_scaled(stack_min(x_packet(mp→xy))) −
              number_to_scaled(stack_max(u_packet(mp→uv))))
          if (number_to_scaled(mp→dely) − mp→tol ≤ number_to_scaled(stack_max(y_packet(mp→xy))) −
                number_to_scaled(stack_min(v_packet(mp→uv))))
            if (number_to_scaled(mp→dely) + mp→tol ≥ number_to_scaled(stack_min(y_packet(mp→xy))) −
                  number_to_scaled(stack_max(v_packet(mp→uv)))) {
              if (number_to_scaled(mp→cur_t) ≥ number_to_scaled(mp→max_t)) {
                if (number_equal(mp→max_t, two_t)) {      /* we've done 17 bisections */
                  set_number_from_scaled(mp→cur_t, ((number_to_scaled(mp→cur_t) + 1)/2));
                  set_number_from_scaled(mp→cur_tt, ((number_to_scaled(mp→cur_tt) + 1)/2));
                  return;
                }
                number_double(mp→max_t);
                number_clone(mp→appr_t, mp→cur_t);
                number_clone(mp→appr_tt, mp→cur_tt);
              }
              ⟨Subdivide for a new level of intersection 618⟩;
              goto CONTINUE;
            }
      if (mp→time_to_go > 0) {
        decr(mp→time_to_go);
      }
      else {
        while (number_less(mp→appr_t, unity_t)) {
          number_double(mp→appr_t);
          number_double(mp→appr_tt);
        }
        number_clone(mp→cur_t, mp→appr_t);
        number_clone(mp→cur_tt, mp→appr_tt);
        return;
      }
    NOT_FOUND:      /* Advance to the next pair (cur_t, cur_tt) */
      if (odd(number_to_scaled(mp→cur_tt))) {
        if (odd(number_to_scaled(mp→cur_t))) {
          /* Descend to the previous level and goto not_found */
          {
            set_number_from_scaled(mp→cur_t, half(number_to_scaled(mp→cur_t)));
            set_number_from_scaled(mp→cur_tt, half(number_to_scaled(mp→cur_tt)));
```

```
          if (number_to_scaled(mp→cur_t) ≡ 0) return;
          mp→bisect_ptr −= int_increment;
          mp→three_l −= (integer) mp→tol_step;
          number_clone(mp→delx, stack_dx);
          number_clone(mp→dely, stack_dy);
          mp→tol = number_to_scaled(stack_tol);
          mp→uv = number_to_scaled(stack_uv);
          mp→xy = number_to_scaled(stack_xy);
          goto NOT_FOUND;
        }
      }
      else {
        set_number_from_scaled(mp→cur_t, number_to_scaled(mp→cur_t) + 1);
        number_add(mp→delx, stack_1(u_packet(mp→uv)));
        number_add(mp→delx, stack_2(u_packet(mp→uv)));
        number_add(mp→delx, stack_3(u_packet(mp→uv)));
        number_add(mp→dely, stack_1(v_packet(mp→uv)));
        number_add(mp→dely, stack_2(v_packet(mp→uv)));
        number_add(mp→dely, stack_3(v_packet(mp→uv)));
        mp→uv = mp→uv + int_packets;      /* switch from l_packets to r_packets */
        set_number_from_scaled(mp→cur_tt, number_to_scaled(mp→cur_tt) − 1);
        mp→xy = mp→xy − int_packets;
        number_add(mp→delx, stack_1(x_packet(mp→xy)));
        number_add(mp→delx, stack_2(x_packet(mp→xy)));
        number_add(mp→delx, stack_3(x_packet(mp→xy)));
        number_add(mp→dely, stack_1(y_packet(mp→xy)));
        number_add(mp→dely, stack_2(y_packet(mp→xy)));
        number_add(mp→dely, stack_3(y_packet(mp→xy)));
      }
    }
    else {
      set_number_from_scaled(mp→cur_tt, number_to_scaled(mp→cur_tt) + 1);
      mp→tol = mp→tol + mp→three_l;
      number_substract(mp→delx, stack_1(x_packet(mp→xy)));
      number_substract(mp→delx, stack_2(x_packet(mp→xy)));
      number_substract(mp→delx, stack_3(x_packet(mp→xy)));
      number_substract(mp→dely, stack_1(y_packet(mp→xy)));
      number_substract(mp→dely, stack_2(y_packet(mp→xy)));
      number_substract(mp→dely, stack_3(y_packet(mp→xy)));
      mp→xy = mp→xy + int_packets;      /* switch from l_packets to r_packets */
    }
  }
}
}
```

614.   The following variables are global, although they are used only by *cubic_intersection*, because it is necessary on some machines to split *cubic_intersection* up into two procedures.

⟨ Global variables 14 ⟩ +≡
  mp_number *delx*;
  mp_number *dely*;   /∗ the components of $\Delta = 2^l(w_0 - z_0)$ ∗/
  integer *tol*;   /∗ bound on the uncertainty in the overlap test ∗/
  integer *uv*;
  integer *xy*;   /∗ pointers to the current packets of interest ∗/
  integer *three_l*;   /∗ *tol_step* times the bisection level ∗/
  mp_number *appr_t*;
  mp_number *appr_tt*;   /∗ best approximations known to the answers ∗/

615.   ⟨ Initialize table entries 182 ⟩ +≡
  new_number(*mp*→*delx*);
  new_number(*mp*→*dely*);
  new_number(*mp*→*appr_t*);
  new_number(*mp*→*appr_tt*);

616.   ⟨ Dealloc variables 27 ⟩ +≡
  free_number(*mp*→*delx*);
  free_number(*mp*→*dely*);
  free_number(*mp*→*appr_t*);
  free_number(*mp*→*appr_tt*);

617.   We shall assume that the coordinates are sufficiently non-extreme that integer overflow will not occur.

⟨ Initialize for intersections at level zero 617 ⟩ ≡

  $q = mp\_next\_knot(p)$;

  $qq = mp\_next\_knot(pp)$;

  $mp{\rightarrow}bisect\_ptr = int\_packets$;

  $set\_number\_from\_substraction(u1r, p{\rightarrow}right\_x, p{\rightarrow}x\_coord)$;

  $set\_number\_from\_substraction(u2r, q{\rightarrow}left\_x, p{\rightarrow}right\_x)$;

  $set\_number\_from\_substraction(u3r, q{\rightarrow}x\_coord, q{\rightarrow}left\_x)$;

  $set\_min\_max(ur\_packet)$;

  $set\_number\_from\_substraction(v1r, p{\rightarrow}right\_y, p{\rightarrow}y\_coord)$;

  $set\_number\_from\_substraction(v2r, q{\rightarrow}left\_y, p{\rightarrow}right\_y)$;

  $set\_number\_from\_substraction(v3r, q{\rightarrow}y\_coord, q{\rightarrow}left\_y)$;

  $set\_min\_max(vr\_packet)$;

  $set\_number\_from\_substraction(x1r, pp{\rightarrow}right\_x, pp{\rightarrow}x\_coord)$;

  $set\_number\_from\_substraction(x2r, qq{\rightarrow}left\_x, pp{\rightarrow}right\_x)$;

  $set\_number\_from\_substraction(x3r, qq{\rightarrow}x\_coord, qq{\rightarrow}left\_x)$;

  $set\_min\_max(xr\_packet)$;

  $set\_number\_from\_substraction(y1r, pp{\rightarrow}right\_y, pp{\rightarrow}y\_coord)$;

  $set\_number\_from\_substraction(y2r, qq{\rightarrow}left\_y, pp{\rightarrow}right\_y)$;

  $set\_number\_from\_substraction(y3r, qq{\rightarrow}y\_coord, qq{\rightarrow}left\_y)$;

  $set\_min\_max(yr\_packet)$;

  $set\_number\_from\_substraction(mp{\rightarrow}delx, p{\rightarrow}x\_coord, pp{\rightarrow}x\_coord)$;

  $set\_number\_from\_substraction(mp{\rightarrow}dely, p{\rightarrow}y\_coord, pp{\rightarrow}y\_coord)$;

  $mp{\rightarrow}tol = 0$;

  $mp{\rightarrow}uv = r\_packets$;

  $mp{\rightarrow}xy = r\_packets$;

  $mp{\rightarrow}three\_l = 0$;

  $set\_number\_from\_scaled(mp{\rightarrow}cur\_t, 1)$; $set\_number\_from\_scaled(mp{\rightarrow}cur\_tt, 1)$

This code is used in section 613.

618.

$\langle$ Subdivide for a new level of intersection $618\,\rangle \equiv$

  $number\_clone\,(stack\_dx,\, mp\text{→}delx\,);$

  $number\_clone\,(stack\_dy,\, mp\text{→}dely\,);$

  $set\_number\_from\_scaled\,(stack\_tol,\, mp\text{→}tol\,);$

  $set\_number\_from\_scaled\,(stack\_uv,\, mp\text{→}uv\,);$

  $set\_number\_from\_scaled\,(stack\_xy,\, mp\text{→}xy\,);$

  $mp\text{→}bisect\_ptr = mp\text{→}bisect\_ptr + int\_increment;$

  $number\_double\,(mp\text{→}cur\_t\,);$

  $number\_double\,(mp\text{→}cur\_tt\,);$

  $number\_clone\,(u1l,\, stack\_1\,(u\_packet\,(mp\text{→}uv\,)));$

  $number\_clone\,(u3r,\, stack\_3\,(u\_packet\,(mp\text{→}uv\,)));$

  $set\_number\_from\_addition\,(u2l,\, u1l,\, stack\_2\,(u\_packet\,(mp\text{→}uv\,)));$

  $number\_half\,(u2l\,);$

  $set\_number\_from\_addition\,(u2r,\, u3r,\, stack\_2\,(u\_packet\,(mp\text{→}uv\,)));$

  $number\_half\,(u2r\,);$

  $set\_number\_from\_addition\,(u3l,\, u2l,\, u2r\,);$

  $number\_half\,(u3l\,);$

  $number\_clone\,(u1r,\, u3l\,);$

  $set\_min\_max\,(ul\_packet\,);$

  $set\_min\_max\,(ur\_packet\,);$

  $number\_clone\,(v1l,\, stack\_1\,(v\_packet\,(mp\text{→}uv\,)));$

  $number\_clone\,(v3r,\, stack\_3\,(v\_packet\,(mp\text{→}uv\,)));$

  $set\_number\_from\_addition\,(v2l,\, v1l,\, stack\_2\,(v\_packet\,(mp\text{→}uv\,)));$

  $number\_half\,(v2l\,);$

  $set\_number\_from\_addition\,(v2r,\, v3r,\, stack\_2\,(v\_packet\,(mp\text{→}uv\,)));$

  $number\_half\,(v2r\,);$

  $set\_number\_from\_addition\,(v3l,\, v2l,\, v2r\,);$

  $number\_half\,(v3l\,);$

  $number\_clone\,(v1r,\, v3l\,);$

  $set\_min\_max\,(vl\_packet\,);$

  $set\_min\_max\,(vr\_packet\,);$

  $number\_clone\,(x1l,\, stack\_1\,(x\_packet\,(mp\text{→}xy\,)));$

  $number\_clone\,(x3r,\, stack\_3\,(x\_packet\,(mp\text{→}xy\,)));$

  $set\_number\_from\_addition\,(x2l,\, x1l,\, stack\_2\,(x\_packet\,(mp\text{→}xy\,)));$

  $number\_half\,(x2l\,);$

  $set\_number\_from\_addition\,(x2r,\, x3r,\, stack\_2\,(x\_packet\,(mp\text{→}xy\,)));$

  $number\_half\,(x2r\,);$

  $set\_number\_from\_addition\,(x3l,\, x2l,\, x2r\,);$

  $number\_half\,(x3l\,);$

  $number\_clone\,(x1r,\, x3l\,);$

  $set\_min\_max\,(xl\_packet\,);$

  $set\_min\_max\,(xr\_packet\,);$

  $number\_clone\,(y1l,\, stack\_1\,(y\_packet\,(mp\text{→}xy\,)));$

  $number\_clone\,(y3r,\, stack\_3\,(y\_packet\,(mp\text{→}xy\,)));$

  $set\_number\_from\_addition\,(y2l,\, y1l,\, stack\_2\,(y\_packet\,(mp\text{→}xy\,)));$

  $number\_half\,(y2l\,);$

  $set\_number\_from\_addition\,(y2r,\, y3r,\, stack\_2\,(y\_packet\,(mp\text{→}xy\,)));$

  $number\_half\,(y2r\,);$

  $set\_number\_from\_addition\,(y3l,\, y2l,\, y2r\,);$

  $number\_half\,(y3l\,);$

  $number\_clone\,(y1r,\, y3l\,);$

$set\_min\_max\,(yl\_packet\,);$
$set\_min\_max\,(yr\_packet\,);$
$mp{\rightarrow}uv = l\_packets\,;$
$mp{\rightarrow}xy = l\_packets\,;$
$number\_double\,(mp{\rightarrow}delx\,);$
$number\_double\,(mp{\rightarrow}dely\,);$
$mp{\rightarrow}tol = mp{\rightarrow}tol - mp{\rightarrow}three\_l + (\textbf{integer})\ mp{\rightarrow}tol\_step\,;$
$mp{\rightarrow}tol\mathrel{+}= mp{\rightarrow}tol\,;\ mp{\rightarrow}three\_l = mp{\rightarrow}three\_l + (\textbf{integer})\ mp{\rightarrow}tol\_step$

This code is used in section 613.

619.   The *path_intersection* procedure is much simpler. It invokes *cubic_intersection* in lexicographic order until finding a pair of cubics that intersect. The final intersection times are placed in *cur_t* and *cur_tt*.

```
static void mp_path_intersection(MP mp, mp_knot h, mp_knot hh)
{
    mp_knot p, pp;      /* link registers that traverse the given paths */
    mp_number n, nn;      /* integer parts of intersection times, minus unity */
    ⟨Change one-point paths into dead cycles 620⟩;
    new_number(n);
    new_number(nn);
    mp→tol_step = 0;
    do {
        set_number_to_unity(n);
        number_negate(n);
        p = h;
        do {
            if (mp_right_type(p) ≠ mp_endpoint) {
                set_number_to_unity(nn);
                number_negate(nn);
                pp = hh;
                do {
                    if (mp_right_type(pp) ≠ mp_endpoint) {
                        mp_cubic_intersection(mp, p, pp);
                        if (number_positive(mp→cur_t)) {
                            number_add(mp→cur_t, n);
                            number_add(mp→cur_tt, nn);
                            goto DONE;
                        }
                    }
                    number_add(nn, unity_t);
                    pp = mp_next_knot(pp);
                } while (pp ≠ hh);
            }
            number_add(n, unity_t);
            p = mp_next_knot(p);
        } while (p ≠ h);
        mp→tol_step = mp→tol_step + 3;
    } while (mp→tol_step ≤ 3);
    number_clone(mp→cur_t, unity_t);
    number_negate(mp→cur_t);
    number_clone(mp→cur_tt, unity_t);
    number_negate(mp→cur_tt);
DONE: free_number(n);
    free_number(nn);
}
```

620.   ⟨Change one-point paths into dead cycles 620⟩ ≡

 if $(\mathit{mp\_right\_type}(h) \equiv \mathit{mp\_endpoint})$ {

  $\mathit{number\_clone}(h{\rightarrow}\mathit{right\_x}, h{\rightarrow}\mathit{x\_coord})$;

  $\mathit{number\_clone}(h{\rightarrow}\mathit{left\_x}, h{\rightarrow}\mathit{x\_coord})$;

  $\mathit{number\_clone}(h{\rightarrow}\mathit{right\_y}, h{\rightarrow}\mathit{y\_coord})$;

  $\mathit{number\_clone}(h{\rightarrow}\mathit{left\_y}, h{\rightarrow}\mathit{y\_coord})$;

  $\mathit{mp\_right\_type}(h) = \mathit{mp\_explicit}$;

 }

 if $(\mathit{mp\_right\_type}(hh) \equiv \mathit{mp\_endpoint})$ {

  $\mathit{number\_clone}(hh{\rightarrow}\mathit{right\_x}, hh{\rightarrow}\mathit{x\_coord})$;

  $\mathit{number\_clone}(hh{\rightarrow}\mathit{left\_x}, hh{\rightarrow}\mathit{x\_coord})$;

  $\mathit{number\_clone}(hh{\rightarrow}\mathit{right\_y}, hh{\rightarrow}\mathit{y\_coord})$;

  $\mathit{number\_clone}(hh{\rightarrow}\mathit{left\_y}, hh{\rightarrow}\mathit{y\_coord})$;

  $\mathit{mp\_right\_type}(hh) = \mathit{mp\_explicit}$;

 }

This code is used in section 619.

621.   Dynamic linear equations.   METAPOST users define variables implicitly by stating equations that should be satisfied; the computer is supposed to be smart enough to solve those equations. And indeed, the computer tries valiantly to do so, by distinguishing five different types of numeric values:

$type(p) = mp\_known$ is the nice case, when $value(p)$ is the *scaled* value of the variable whose address is p.

$type(p) = mp\_dependent$ means that $value(p)$ is not present, but $dep\_list(p)$ points to a *dependency list* that expresses the value of variable p as a *scaled* number plus a sum of independent variables with *fraction* coefficients.

$type(p) = mp\_independent$ means that $indep\_value(p) = s$, where $s > 0$ is a "serial number" reflecting the time this variable was first used in an equation; and there is an extra field $indep\_scale(p) = m$, with $0 \leq m < 64$, each dependent variable that refers to this one is actually referring to the future value of this variable times 2^m. (Usually $m = 0$, but higher degrees of scaling are sometimes needed to keep the coefficients in dependency lists from getting too large. The value of m will always be even.)

$type(p) = mp\_numeric\_type$ means that variable p hasn't appeared in an equation before, but it has been explicitly declared to be numeric.

$type(p) = undefined$ means that variable p hasn't appeared before.

We have actually discussed these five types in the reverse order of their history during a computation: Once *known*, a variable never again becomes *dependent*; once *dependent*, it almost never again becomes *mp_independent*; once *mp_independent*, it never again becomes *mp_numeric_type*; and once *mp_numeric_type*, it never again becomes *undefined* (except of course when the user specifically decides to scrap the old value and start again). A backward step may, however, take place: Sometimes a *dependent* variable becomes *mp_independent* again, when one of the independent variables it depends on is reverting to *undefined*.

#define $indep\_scale(A)$ $((\mathbf{mp\_value\_node})(A))\rightarrow data.indep.scale$
#define $set\_indep\_scale(A, B)$ $((\mathbf{mp\_value\_node})(A))\rightarrow data.indep.scale = (B)$
#define $indep\_value(A)$ $((\mathbf{mp\_value\_node})(A))\rightarrow data.indep.serial$
#define $set\_indep\_value(A, B)$ $((\mathbf{mp\_value\_node})(A))\rightarrow data.indep.serial = (B)$

```
  void mp_new_indep(MP mp, mp_node p)
  {    /* create a new independent variable */
    if (mp→serial_no ≥ max_integer) {
      mp_fatal_error(mp, "variable␣instance␣identifiers␣exhausted");
    }
    mp_type(p) = mp_independent;
    mp→serial_no = mp→serial_no + 1;
    set_indep_scale(p, 0);
    set_indep_value(p, mp→serial_no);
  }
```

622.   ⟨ Declarations 8 ⟩ +≡
  void $mp\_new\_indep(\mathbf{MP}\ mp, \mathbf{mp\_node}\ p)$;

623.   ⟨ Global variables 14 ⟩ +≡
  integer $serial\_no$;   /* the most recent serial number */

624.   But how are dependency lists represented? It's simple: The linear combination $\alpha_1 v_1 + \cdots + \alpha_k v_k + \beta$ appears in $k + 1$ value nodes. If $q = dep\_list(p)$ points to this list, and if $k > 0$, then $dep\_value(q) = \alpha_1$ (which is a *fraction*); $dep\_info(q)$ points to the location of α_1; and $mp\_link(p)$ points to the dependency list $\alpha_2 v_2 + \cdots + \alpha_k v_k + \beta$. On the other hand if $k = 0$, then $dep\_value(q) = \beta$ (which is *scaled*) and $dep\_info(q) = \Lambda$. The independent variables $v_1, \ldots, v_k$ have been sorted so that they appear in decreasing order of their *value* fields (i.e., of their serial numbers). (It is convenient to use decreasing order, since $value(\Lambda) = 0$. If the independent variables were not sorted by serial number but by some other criterion, such as their location in *mem*, the equation-solving mechanism would be too system-dependent, because the ordering can affect the computed results.)

The *link* field in the node that contains the constant term β is called the *final link* of the dependency list. METAPOST maintains a doubly-linked master list of all dependency lists, in terms of a permanently allocated node in *mem* called *dep_head*. If there are no dependencies, we have $mp\_link(dep\_head) = dep\_head$ and $prev\_dep(dep\_head) = dep\_head$; otherwise $mp\_link(dep\_head)$ points to the first dependent variable, say p, and $prev\_dep(p) = dep\_head$. We have $type(p) = mp\_dependent$, and $dep\_list(p)$ points to its dependency list. If the final link of that dependency list occurs in location q, then $mp\_link(q)$ points to the next dependent variable (say r); and we have $prev\_dep(r) = q$, etc.

Dependency nodes sometimes mutate into value nodes and vice versa, so their structures have to match.

```
#define  dep_value(A)  ((mp_value_node)(A))⃗data.n
#define  set_dep_value(A, B)  do_set_dep_value(mp, (A), (B))
#define  dep_info(A)  get_dep_info(mp, (A))
#define  set_dep_info(A, B)  do
        {
          mp_value_node d = (mp_value_node)(B);
          FUNCTION_TRACE4("set_dep_info(%p,%p) on %d\n", (A), d, __LINE__);
          ((mp_value_node)(A))⃗parent_ = (mp_node) d;
        }
        while (0)
#define  dep_list(A)  ((mp_value_node)(A))⃗attr_head_
          /* half of the value field in a dependent variable */
#define  set_dep_list(A, B)  do
        {
          mp_value_node d = (mp_value_node)(B);
          FUNCTION_TRACE4("set_dep_list(%p,%p) on %d\n", (A), d, __LINE__);
          dep_list((A)) = (mp_node) d;
        }
        while (0)
#define  prev_dep(A)  ((mp_value_node)(A))⃗subscr_head_
          /* the other half; makes a doubly linked list */
#define  set_prev_dep(A, B)  do
        {
          mp_value_node d = (mp_value_node)(B);
          FUNCTION_TRACE4("set_prev_dep(%p,%p) on %d\n", (A), d, __LINE__);
          prev_dep((A)) = (mp_node) d;
        }
        while (0)
  static mp_node get_dep_info(MP mp, mp_value_node p)
  {
    mp_node d;
    d = p⃗parent_;      /* half of the value field in a dependent variable */
    FUNCTION_TRACE3("%p = dep_info(%p)\n", d, p);
    return d;
```

```
}
static void do_set_dep_value(MP mp, mp_value_node p, mp_number q)
{
    number_clone(p⃗data.n, q);     /* half of the value field in a dependent variable */
    FUNCTION_TRACE3("set_dep_value(%p,%d)\n", p, q);
    p⃗attr_head_ = Λ;
    p⃗subscr_head_ = Λ;
}
```

625. ⟨Declarations 8⟩ +≡
```
static mp_node get_dep_info(MP mp, mp_value_node p);
```

626.
```
static mp_value_node mp_get_dep_node(MP mp)
{
    mp_value_node p = (mp_value_node) mp_get_value_node(mp);
    mp_type(p) = mp_dep_node_type;
    return p;
}
static void mp_free_dep_node(MP mp, mp_value_node p)
{
    mp_free_value_node(mp, (mp_node) p);
}
```

627. ⟨Declarations 8⟩ +≡
```
static void mp_free_dep_node(MP mp, mp_value_node p);
```

628. ⟨Initialize table entries 182⟩ +≡
```
mp⃗serial_no = 0;
mp⃗dep_head = mp_get_dep_node(mp);
set_mp_link(mp⃗dep_head, (mp_node) mp⃗dep_head);
set_prev_dep(mp⃗dep_head, (mp_node) mp⃗dep_head);
set_dep_info(mp⃗dep_head, Λ);
set_dep_list(mp⃗dep_head, Λ);
```

629. ⟨Free table entries 183⟩ +≡
```
mp_free_dep_node(mp, mp⃗dep_head);
```

630. Actually the description above contains a little white lie. There's another kind of variable called *mp_proto_dependent*, which is just like a *dependent* one except that the α coefficients in its dependency list are *scaled* instead of being fractions. Proto-dependency lists are mixed with dependency lists in the nodes reachable from *dep_head*.

631. Here is a procedure that prints a dependency list in symbolic form. The second parameter should be either *dependent* or *mp_proto_dependent*, to indicate the scaling of the coefficients.

⟨Declarations 8⟩ +≡
```
static void mp_print_dependency(MP mp, mp_value_node p, quarterword t);
```

632.   **void** $mp\_print\_dependency(\textbf{MP}\ mp, \textbf{mp\_value\_node}\ p, \textbf{quarterword}\ t)$
```
{
```
 mp_number v; /∗ a coefficient ∗/
 mp_value_node pp; /∗ for list manipulation ∗/
 mp_node q;

 $pp = p$;
 $new\_number(v)$;
 while $(true)$ {
  $number\_clone(v, dep\_value(p))$;
  $number\_abs(v)$;
  $q = dep\_info(p)$;
  if $(q \equiv \Lambda)$ { /∗ the constant term ∗/
   if $(number\_nonzero(v) \vee (p \equiv pp))$ {
    if $(number\_positive(dep\_value(p)))$
     if $(p \neq pp)$ $mp\_print\_char(mp, xord('+'))$;
    $print\_number(dep\_value(p))$;
   }
   return;
  } /∗ Print the coefficient, unless it's ±1.0 ∗/
  if $(number\_negative(dep\_value(p)))$ $mp\_print\_char(mp, xord('-'))$;
  else if $(p \neq pp)$ $mp\_print\_char(mp, xord('+'))$;
  if $(t \equiv mp\_dependent)$ {
   $fraction\_to\_round\_scaled(v)$;
  }
  if $(\neg number\_equal(v, unity\_t))$ $print\_number(v)$;
  if $(mp\_type(q) \neq mp\_independent)$ $mp\_confusion(mp, \texttt{"dep"})$;
  $mp\_print\_variable\_name(mp, q)$;
  $set\_number\_from\_scaled(v, indep\_scale(q))$;
  while $(number\_positive(v))$ {
   $mp\_print(mp, \texttt{"*4"})$;
   $number\_add\_scaled(v, -2)$;
  }
  $p = (\textbf{mp\_value\_node})\ mp\_link(p)$;
 }
```
}
```

633.   The maximum absolute value of a coefficient in a given dependency list is returned by the following simple function.

> **static void** mp_max_coef (**MP** mp, **mp_number** $*x$, **mp_value_node** p)
> {
>   **mp_number**($absv$);
> new_number($absv$);
> $set\_number\_to\_zero$($*x$);
>   **while** (dep_info(p) $\neq \Lambda$) {
> $number\_clone$($absv$, dep_value(p));
> $number\_abs$($absv$);
>   **if** ($number\_greater$($absv$, $*x$)) {
> $number\_clone$($*x$, $absv$);
>   }
> $p =$ (**mp_value_node**) mp_link(p);
>   }
> $free\_number$($absv$);
> }

634.   One of the main operations needed on dependency lists is to add a multiple of one list to the other; we call this p_plus_fq, where p and q point to dependency lists and f is a fraction.

If the coefficient of any independent variable becomes $coef\_bound$ or more, in absolute value, this procedure changes the type of that variable to '$independent\_needing\_fix$', and sets the global variable fix_needed to $true$. The value of $coef\_bound = \mu$ is chosen so that $\mu^2 + \mu < 8$; this means that the numbers we deal with won't get too large. (Instead of the "optimum" $\mu = (\sqrt{33} - 1)/2 \approx 2.3723$, the safer value $7/3$ is taken as the threshold.)

The changes mentioned in the preceding paragraph are actually done only if the global variable $watch\_coefs$ is $true$. But it usually is; in fact, it is $false$ only when METAPOST is making a dependency list that will soon be equated to zero.

Several procedures that act on dependency lists, including p_plus_fq, set the global variable dep_final to the final (constant term) node of the dependency list that they produce.

#define $independent\_needing\_fix$ 0

⟨ Global variables 14 ⟩ +≡
>   **boolean** fix_needed;   /∗ does at least one $independent$ variable need scaling? ∗/
>   **boolean** $watch\_coefs$;   /∗ should we scale coefficients that exceed $coef\_bound$? ∗/
>   **mp_value_node** dep_final;   /∗ location of the constant term and final link ∗/

635.   ⟨ Set initial values of key variables 38 ⟩ +≡
> $mp\text{→}fix\_needed = false$;
> $mp\text{→}watch\_coefs = true$;

636.   The *p_plus_fq* procedure has a fourth parameter, *t*, that should be set to *mp_proto_dependent* if *p* is a proto-dependency list. In this case *f* will be *scaled*, not a *fraction*. Similarly, the fifth parameter *tt* should be *mp_proto_dependent* if *q* is a proto-dependency list.

List *q* is unchanged by the operation; but list *p* is totally destroyed.

The final link of the dependency list or proto-dependency list returned by *p_plus_fq* is the same as the original final link of *p*. Indeed, the constant term of the result will be located in the same *mem* location as the original constant term of *p*.

Coefficients of the result are assumed to be zero if they are less than a certain threshold. This compensates for inevitable rounding errors, and tends to make more variables '*known*'. The threshold is approximately 10^{-5} in the case of normal dependency lists, 10^{-4} for proto-dependencies.

#define   *fraction_threshold_k*   ((**math_data** ∗) *mp→math*)→*fraction_threshold_t*
#define   *half_fraction_threshold_k*   ((**math_data** ∗) *mp→math*)→*half_fraction_threshold_t*
#define   *scaled_threshold_k*   ((**math_data** ∗) *mp→math*)→*scaled_threshold_t*
#define   *half_scaled_threshold_k*   ((**math_data** ∗) *mp→math*)→*half_scaled_threshold_t*

⟨ Declarations 8 ⟩ +≡
    static mp_value_node *mp_p_plus_fq*(**MP** *mp*, **mp_value_node** *p*, **mp_number** *f*, **mp_value_node**
        q, *mp_variable_type t*, *mp_variable_type tt*);

637.   static **mp_value_node** $mp\_p\_plus\_fq$(**MP** mp, **mp_value_node** p, **mp_number** f, **mp_value_node** q, $mp\_variable\_type\,t$, $mp\_variable\_type\,tt$)
{
    mp_node pp, qq;   /∗ $dep\_info(p)$ and $dep\_info(q)$, respectively ∗/
    mp_value_node r, s;   /∗ for list manipulation ∗/
    mp_number $threshold$, $half\_threshold$;   /∗ defines a neighborhood of zero ∗/
    mp_number v, vv;   /∗ temporary registers ∗/
    $new\_number(v)$;
    $new\_number(vv)$;
    $new\_number(threshold)$;
    $new\_number(half\_threshold)$;
    if ($t \equiv mp\_dependent$) {
       $number\_clone(threshold, fraction\_threshold\_k)$;
       $number\_clone(half\_threshold, half\_fraction\_threshold\_k)$;
    }
    else {
       $number\_clone(threshold, scaled\_threshold\_k)$;
       $number\_clone(half\_threshold, half\_scaled\_threshold\_k)$;
    }
    $r = ($**mp_value_node**$) mp\rightarrow temp\_head$;
    $pp = dep\_info(p)$;
    $qq = dep\_info(q)$;
    while (1) {
      if ($pp \equiv qq$) {
        if ($pp \equiv \Lambda$) {
          break;
        }
        else {   /∗ Contribute a term from p, plus f times the corresponding term from q ∗/
          mp_number $r1$;
          mp_number $absv$;
          $new\_fraction(r1)$;
          $new\_number(absv)$;
          if ($tt \equiv mp\_dependent$) {
            $take\_fraction(r1, f, dep\_value(q))$;
          }
          else {
            $take\_scaled(r1, f, dep\_value(q))$;
          }
          $set\_number\_from\_addition(v, dep\_value(p), r1)$;
          $free\_number(r1)$;
          $set\_dep\_value(p, v)$;
          $s = p$;
          $p = ($**mp_value_node**$) mp\_link(p)$;
          $number\_clone(absv, v)$;
          $number\_abs(absv)$;
          if ($number\_less(absv, threshold)$) {
            $mp\_free\_dep\_node(mp, s)$;
          }
          else {
            if ($number\_greaterequal(absv, coef\_bound\_k) \wedge mp\rightarrow watch\_coefs$) {
              $mp\_type(qq) = independent\_needing\_fix$;
              $mp\rightarrow fix\_needed = true$;

```
        }
        set_mp_link(r, (mp_node) s);
        r = s;
      }
      free_number(absv);
      pp = dep_info(p);
      q = (mp_value_node) mp_link(q);
      qq = dep_info(q);
    }
  }
  else {
    if (pp ≡ Λ) set_number_to_neg_inf(v);
    else if (mp_type(pp) ≡ mp_independent) set_number_from_scaled(v, indep_value(pp));
    else number_clone(v, value_number(pp));
    if (qq ≡ Λ) set_number_to_neg_inf(vv);
    else if (mp_type(qq) ≡ mp_independent) set_number_from_scaled(vv, indep_value(qq));
    else number_clone(vv, value_number(qq));
    if (number_less(v, vv)) {        /* Contribute a term from q, multiplied by f */
      mp_number absv;

      new_number(absv);
      {
        mp_number r1;
        mp_number arg1, arg2;

        new_fraction(r1);
        new_number(arg1);
        new_number(arg2);
        number_clone(arg1, f);
        number_clone(arg2, dep_value(q));
        if (tt ≡ mp_dependent) {
          take_fraction(r1, arg1, arg2);
        }
        else {
          take_scaled(r1, arg1, arg2);
        }
        number_clone(v, r1);
        free_number(r1);
        free_number(arg1);
        free_number(arg2);
      }
      number_clone(absv, v);
      number_abs(absv);
      if (number_greater(absv, half_threshold)) {
        s = mp_get_dep_node(mp);
        set_dep_info(s, qq);
        set_dep_value(s, v);
        if (number_greaterequal(absv, coef_bound_k) ∧ mp→watch_coefs) {
            /* clang: dereference of a null pointer ('qq') */
          assert(qq);
          mp_type(qq) = independent_needing_fix;
          mp→fix_needed = true;
        }
        set_mp_link(r, (mp_node) s);
```

```
          r = s;
        }
        q = (mp_value_node) mp_link(q);
        qq = dep_info(q);
        free_number(absv);
      }
      else {
        set_mp_link(r, (mp_node) p);
        r = p;
        p = (mp_value_node) mp_link(p);
        pp = dep_info(p);
      }
    }
  }
  {
    mp_number r1;
    mp_number arg1, arg2;

    new_fraction(r1);
    new_number(arg1);
    new_number(arg2);
    number_clone(arg1, dep_value(q));
    number_clone(arg2, f);
    if (t ≡ mp_dependent) {
      take_fraction(r1, arg1, arg2);
    }
    else {
      take_scaled(r1, arg1, arg2);
    }
    slow_add(arg1, dep_value(p), r1);
    set_dep_value(p, arg1);
    free_number(r1);
    free_number(arg1);
    free_number(arg2);
  }
  set_mp_link(r, (mp_node) p);
  mp→dep_final = p;
  free_number(threshold);
  free_number(half_threshold);
  free_number(v);
  free_number(vv);
  return (mp_value_node) mp_link(mp→temp_head);
}
```

638.   It is convenient to have another subroutine for the special case of *p_plus_fq* when $f = 1.0$. In this routine lists p and q are both of the same type t (either *dependent* or *mp_proto_dependent*).

```
static mp_value_node mp_p_plus_q(MP mp, mp_value_node p, mp_value_node q, mp_variable_typet)
{
    mp_node pp, qq;      /* dep_info(p) and dep_info(q), respectively */
    mp_value_node s;     /* for list manipulation */
    mp_value_node r;     /* for list manipulation */
    mp_number threshold;     /* defines a neighborhood of zero */
    mp_number v, vv;     /* temporary register */
    new_number(v);
    new_number(vv);
    new_number(threshold);
    if (t ≡ mp_dependent) number_clone(threshold, fraction_threshold_k);
    else number_clone(threshold, scaled_threshold_k);
    r = (mp_value_node) mp→temp_head;
    pp = dep_info(p);
    qq = dep_info(q);
    while (1) {
        if (pp ≡ qq) {
            if (pp ≡ Λ) {
                break;
            }
            else {     /* Contribute a term from p, plus the corresponding term from q */
                mp_number test;

                new_number(test);
                set_number_from_addition(v, dep_value(p), dep_value(q));
                set_dep_value(p, v);
                s = p;
                p = (mp_value_node) mp_link(p);
                pp = dep_info(p);
                number_clone(test, v);
                number_abs(test);
                if (number_less(test, threshold)) {
                    mp_free_dep_node(mp, s);
                }
                else {
                    if (number_greaterequal(test, coef_bound_k) ∧ mp→watch_coefs) {
                        mp_type(qq) = independent_needing_fix;
                        mp→fix_needed = true;
                    }
                    set_mp_link(r, (mp_node) s);
                    r = s;
                }
                free_number(test);
                q = (mp_value_node) mp_link(q);
                qq = dep_info(q);
            }
        }
        else {
            if (pp ≡ Λ) set_number_to_zero(v);
            else if (mp_type(pp) ≡ mp_independent) set_number_from_scaled(v, indep_value(pp));
```

```
          else  number_clone(v, value_number(pp));
          if  (qq ≡ Λ)  set_number_to_zero(vv);
          else if  (mp_type(qq) ≡ mp_independent)  set_number_from_scaled(vv, indep_value(qq));
          else  number_clone(vv, value_number(qq));
          if  (number_less(v, vv))  {
             s = mp_get_dep_node(mp);
             set_dep_info(s, qq);
             set_dep_value(s, dep_value(q));
             q = (mp_value_node)  mp_link(q);
             qq = dep_info(q);
             set_mp_link(r, (mp_node) s);
             r = s;
          }
          else  {
             set_mp_link(r, (mp_node) p);
             r = p;
             p = (mp_value_node)  mp_link(p);
             pp = dep_info(p);
          }
       }
    }
    {
       mp_number  r1;

       new_number(r1);
       slow_add(r1, dep_value(p), dep_value(q));
       set_dep_value(p, r1);
       free_number(r1);
    }
    set_mp_link(r, (mp_node) p);
    mp→dep_final = p;
    free_number(v);
    free_number(vv);
    free_number(threshold);
    return (mp_value_node)  mp_link(mp→temp_head);
 }
```

639.   A somewhat simpler routine will multiply a dependency list by a given constant v. The constant is
either a *fraction* less than *fraction_one*, or it is *scaled*. In the latter case we might be forced to convert a
dependency list to a proto-dependency list. Parameters *t0* and *t1* are the list types before and after; they
should agree unless $t0 = mp\_dependent$ and $t1 = mp\_proto\_dependent$ and $v\_is\_scaled = true$.

> **static mp_value_node** $mp\_p\_times\_v$(**MP** mp, **mp_value_node** p, **mp_number** v, **quarterword**
> $t0$, **quarterword** $t1$, **boolean** v_is_scaled)
> {
>   **mp_value_node** r, s;   /∗ for list manipulation ∗/
>   **mp_number** w;   /∗ tentative coefficient ∗/
>   **mp_number** $threshold$;
>   **boolean** $scaling\_down$;
>
> $new\_number(threshold)$;
> $new\_number(w)$;
>   **if** $(t0 \neq t1)$ $scaling\_down = true$;
>   **else** $scaling\_down = (\neg v\_is\_scaled)$;
>   **if** $(t1 \equiv mp\_dependent)$ $number\_clone(threshold, half\_fraction\_threshold\_k)$;
>   **else** $number\_clone(threshold, half\_scaled\_threshold\_k)$;
> $r = (\textbf{mp\_value\_node})\ mp\text{-}temp\_head$;
>   **while** $(dep\_info(p) \neq \Lambda)$ {
>   **mp_number** $test$;
>
> $new\_number(test)$;
>   **if** $(scaling\_down)$ {
> $take\_fraction(w, v, dep\_value(p))$;
>   }
>   **else** {
> $take\_scaled(w, v, dep\_value(p))$;
>   }
> $number\_clone(test, w)$;
> $number\_abs(test)$;
>   **if** $(number\_lessequal(test, threshold))$ {
> $s = (\textbf{mp\_value\_node})\ mp\_link(p)$;
> $mp\_free\_dep\_node(mp, p)$;
> $p = s$;
>   }
>   **else** {
>   **if** $(number\_greaterequal(test, coef\_bound\_k))$ {
> $mp\text{-}fix\_needed = true$;
> $mp\_type(dep\_info(p)) = independent\_needing\_fix$;
>   }
> $set\_mp\_link(r, (\textbf{mp\_node})\ p)$;
> $r = p$;
> $set\_dep\_value(p, w)$;
> $p = (\textbf{mp\_value\_node})\ mp\_link(p)$;
>   }
> $free\_number(test)$;
>   }
> $set\_mp\_link(r, (\textbf{mp\_node})\ p)$;
>   {
>   **mp_number** $r1$;
>
> $new\_number(r1)$;
>   **if** (v_is_scaled) {

$take\_scaled(r1, dep\_value(p), v);$

     }

   else {

$take\_fraction(r1, dep\_value(p), v);$

     }

$set\_dep\_value(p, r1);$

$free\_number(r1);$

   }

$free\_number(w);$

$free\_number(threshold);$

return (**mp_value_node**) $mp\_link(mp\rightarrow temp\_head);$

}

640.   Similarly, we sometimes need to divide a dependency list by a given *scaled* constant.

⟨ Declarations 8 ⟩ +≡

  static mp_value_node $mp\_p\_over\_v($**MP** $mp,$ **mp_value_node** $p,$ **mp_number** $v,$ **quarterword**
$t0,$ **quarterword** $t1);$

641.

#define $p\_over\_v\_threshold\_k$ ((**math_data** ∗) $mp\rightarrow math)\rightarrow p\_over\_v\_threshold\_t$

  mp_value_node $mp\_p\_over\_v$(**MP** mp, **mp_value_node** p, **mp_number** v_orig, **quarterword**
      $t0$, **quarterword** $t1$)

 {

   mp_value_node r, s;   /∗ for list manipulation ∗/
   mp_number w;   /∗ tentative coefficient ∗/
   mp_number $threshold$;
   mp_number v;
   boolean $scaling\_down$;

   $new\_number(v)$;
   $new\_number(w)$;
   $new\_number(threshold)$;
   $number\_clone(v, v\_orig)$;
   if $(t0 \neq t1)$ $scaling\_down = true$;
   else $scaling\_down = false$;
   if $(t1 \equiv mp\_dependent)$ $number\_clone(threshold, half\_fraction\_threshold\_k)$;
   else $number\_clone(threshold, half\_scaled\_threshold\_k)$;
   $r = ($**mp_value_node**$)$ $mp\rightarrow temp\_head$;
   while $(dep\_info(p) \neq \Lambda)$ {
     if $(scaling\_down)$ {
       mp_number x, $absv$;

       $new\_number(x)$;
       $new\_number(absv)$;
       $number\_clone(absv, v)$;
       $number\_abs(absv)$;
       if $(number\_less(absv, p\_over\_v\_threshold\_k))$ {
         $number\_clone(x, v)$;
         $convert\_scaled\_to\_fraction(x)$;
         $make\_scaled(w, dep\_value(p), x)$;
       }
       else {
         $number\_clone(x, dep\_value(p))$;
         $fraction\_to\_round\_scaled(x)$;
         $make\_scaled(w, x, v)$;
       }
       $free\_number(x)$;
       $free\_number(absv)$;
     }
     else {
       $make\_scaled(w, dep\_value(p), v)$;
     }
     {
       mp_number $test$;

       $new\_number(test)$;
       $number\_clone(test, w)$;
       $number\_abs(test)$;
       if $(number\_lessequal(test, threshold))$ {
         $s = ($**mp_value_node**$)$ $mp\_link(p)$;
         $mp\_free\_dep\_node(mp, p)$;
         $p = s$;

```
      }
      else  {
        if  (number_greaterequal(test, coef_bound_k))  {
          mp→fix_needed = true;
          mp_type(dep_info(p)) = independent_needing_fix;
        }
        set_mp_link(r, (mp_node) p);
        r = p;
        set_dep_value(p, w);
        p = (mp_value_node) mp_link(p);
      }
      free_number(test);
    }
  }
  set_mp_link(r, (mp_node) p);
  {
    mp_number  ret;

    new_number(ret);
    make_scaled(ret, dep_value(p), v);
    set_dep_value(p, ret);
    free_number(ret);
  }
  free_number(v);
  free_number(w);
  free_number(threshold);
  return (mp_value_node) mp_link(mp→temp_head);
}
```

642.   Here's another utility routine for dependency lists. When an independent variable becomes dependent, we want to remove it from all existing dependencies. The $p\_with\_x\_becoming\_q$ function computes the dependency list of p after variable x has been replaced by q.

This procedure has basically the same calling conventions as p_plus_fq: List q is unchanged; list p is destroyed; the constant node and the final link are inherited from p; and the fourth parameter tells whether or not p is $mp\_proto\_dependent$. However, the global variable dep_final is not altered if x does not occur in list p.

```
static mp_value_node mp_p_with_x_becoming_q(MP mp, mp_value_node p, mp_node x, mp_node
        q, quarterword t)
{
  mp_value_node r, s;     /* for list manipulation */
  integer sx;     /* serial number of x */

  s = p;
  r = (mp_value_node) mp→temp_head;
  sx = indep_value(x);
  while (dep_info(s) ≠ Λ ∧ indep_value(dep_info(s)) > sx) {
    r = s;
    s = (mp_value_node) mp_link(s);
  }
  if (dep_info(s) ≡ Λ ∨ dep_info(s) ≠ x) {
    return p;
  }
  else {
    mp_value_node ret;
    mp_number v1;

    new_number(v1);
    set_mp_link(mp→temp_head, (mp_node) p);
    set_mp_link(r, mp_link(s));
    number_clone(v1, dep_value(s));
    mp_free_dep_node(mp, s);
    ret = mp_p_plus_fq(mp, (mp_value_node) mp_link(mp→temp_head), v1, (mp_value_node)
        q, t, mp_dependent);
    free_number(v1);
    return ret;
  }
}
```

643.   Here's a simple procedure that reports an error when a variable has just received a known value that's out of the required range.

⟨ Declarations 8 ⟩ +≡
```
  static void mp_val_too_big(MP mp, mp_number x);
```

644.   **static void** $mp\_val\_too\_big(\textbf{MP}\ mp, \textbf{mp\_number}\ x)$
$\{$
   if $(number\_positive(internal\_value(mp\_warning\_check)))\ \{$
      char $msg[256];$
      const char $*hlp[\,] = \{$"The␣equation␣I␣just␣processed␣has␣given␣some␣variable␣a",
         "value␣outside␣of␣the␣safetyp␣range.␣Continue␣and␣I'll␣try",
         "to␣cope␣with␣that␣big␣value;␣but␣it␣might␣be␣dangerous.",
         "(Set␣warningcheck:=0␣to␣suppress␣this␣message.)", $\Lambda\};$
      $mp_snprintf(msg, 256, $"Value␣is␣too␣large␣(%s)", $number\_tostring(x));$
      $mp\_error(mp, msg, hlp, true);$
   $\}$
$\}$

645.   When a dependent variable becomes known, the following routine removes its dependency list. Here p points to the variable, and q points to the dependency list (which is one node long).

$\langle$ Declarations $8\,\rangle\,+\equiv$
   static void $mp\_make\_known(\textbf{MP}\ mp, \textbf{mp\_value\_node}\ p, \textbf{mp\_value\_node}\ q);$

646.   **void** $mp\_make\_known(\textbf{MP}\ mp, \textbf{mp\_value\_node}\ p, \textbf{mp\_value\_node}\ q)$
$\{$
   $mp\_variable\_type\ t;$ /∗ the previous type ∗/
   mp_number $absp;$
   $new\_number(absp);$
   $set\_prev\_dep(mp\_link(q), prev\_dep(p));$
   $set\_mp\_link(prev\_dep(p), mp\_link(q));$
   $t = mp\_type(p);$
   $mp\_type(p) = mp\_known;$
   $set\_value\_number(p, dep\_value(q));$
   $mp\_free\_dep\_node(mp, q);$
   $number\_clone(absp, value\_number(p));$
   $number\_abs(absp);$
   if $(number\_greaterequal(absp, warning\_limit\_t))\ \ mp\_val\_too\_big(mp, value\_number(p));$
   if $((number\_positive(internal\_value(mp\_tracing\_equations))) \wedge mp\_interesting(mp, (\textbf{mp\_node})\ p))\ \{$
      $mp\_begin\_diagnostic(mp);$
      $mp_print_nl(mp, $"####␣");
      $mp\_print\_variable\_name(mp, (\textbf{mp\_node})\ p);$
      $mp\_print\_char(mp, xord($'='$));$
      $print\_number(value\_number(p));$
      $mp\_end\_diagnostic(mp, false);$
   $\}$
   if $(cur\_exp\_node(\,) \equiv (\textbf{mp\_node})\ p \wedge mp\text{-}cur\_exp.type \equiv t)\ \{$
      $mp\text{-}cur\_exp.type = mp\_known;$
      $set\_cur\_exp\_value\_number(value\_number(p));$
      $mp\_free\_value\_node(mp, (\textbf{mp\_node})\ p);$
   $\}$
   $free\_number(absp);$
$\}$

647.   The *fix_dependencies* routine is called into action when *fix_needed* has been triggered. The program keeps a list s of independent variables whose coefficients must be divided by 4.

In unusual cases, this fixup process might reduce one or more coefficients to zero, so that a variable will become known more or less by default.

⟨ Declarations 8 ⟩ +≡
  static void *mp_fix_dependencies*(**MP** *mp*);

648.

#define *independent_being_fixed* 1   /∗ this variable already appears in *s* ∗/

  static void *mp_fix_dependencies*(**MP** *mp*)
  {
    mp_value_node *p*, *q*, *r*, *s*, *t*;   /∗ list manipulation registers ∗/
    mp_node *x*;   /∗ an independent variable ∗/
    r = (**mp_value_node**) *mp_link*(*mp*→*dep_head*);
    s = Λ;
    while (*r* ≠ *mp*→*dep_head*) {
      t = *r*;
      /∗ Run through the dependency list for variable *t*, fixing all nodes, and ending with final link *q* ∗/
      while (1) {
        if (*t* ≡ *r*) {
          q = (**mp_value_node**) *dep_list*(*t*);
        }
        else {
          q = (**mp_value_node**) *mp_link*(*r*);
        }
        x = *dep_info*(*q*);
        if (*x* ≡ Λ) **break**;
        if (*mp_type*(*x*) ≤ *independent_being_fixed*) {
          if (*mp_type*(*x*) < *independent_being_fixed*) {
            p = *mp_get_dep_node*(*mp*);
            set_mp_link(*p*, (**mp_node**) *s*);
            s = *p*;
            set_dep_info(*s*, *x*);
            mp_type(*x*) = *independent_being_fixed*;
          }
          set_dep_value(*q*, *dep_value*(*q*));
          number_divide_int(*dep_value*(*q*), 4);
          if (*number_zero*(*dep_value*(*q*))) {
            set_mp_link(*r*, *mp_link*(*q*));
            mp_free_dep_node(*mp*, *q*);
            q = *r*;
          }
        }
        r = *q*;
      }
      r = (**mp_value_node**) *mp_link*(*q*);
      if (*q* ≡ (**mp_value_node**) *dep_list*(*t*)) *mp_make_known*(*mp*, *t*, *q*);
    }
    while (*s* ≠ Λ) {
      p = (**mp_value_node**) *mp_link*(*s*);
      x = *dep_info*(*s*);
      mp_free_dep_node(*mp*, *s*);
      s = *p*;
      mp_type(*x*) = *mp_independent*;
      set_indep_scale(*x*, *indep_scale*(*x*) + 2);
    }
    mp→*fix_needed* = *false*;
  }

649.   The *new_dep* routine installs a dependency list p based on the value node q, linking it into the list of all known dependencies. It replaces q with the new dependency node. We assume that *dep_final* points to the final node of list p.

> **static void** mp_new_dep(**MP** mp, **mp_node** q, $mp\_variable\_type\,newtype$, **mp_value_node** p)
> {
>   **mp_node** r;   /∗ what used to be the first dependency ∗/
>   FUNCTION_TRACE4("mp_new_dep(%p,%d,%p)\n", q, $newtype$, p);
> $mp\_type(q) = newtype$;
> $set\_dep\_list(q, p)$;
> $set\_prev\_dep(q, ($**mp_node**$)\ mp{\rightarrow}dep\_head)$;
> $r = mp\_link(mp{\rightarrow}dep\_head)$;
> $set\_mp\_link(mp{\rightarrow}dep\_final, r)$;
> $set\_prev\_dep(r, ($**mp_node**$)\ mp{\rightarrow}dep\_final)$;
> $set\_mp\_link(mp{\rightarrow}dep\_head, q)$;
> }

650.   Here is one of the ways a dependency list gets started. The *const_dependency* routine produces a list that has nothing but a constant term.

> **static mp_value_node** $mp\_const\_dependency$(**MP** mp, **mp_number** v)
> {
> $mp{\rightarrow}dep\_final = mp\_get\_dep\_node(mp)$;
> $set\_dep\_value(mp{\rightarrow}dep\_final, v)$;
> $set\_dep\_info(mp{\rightarrow}dep\_final, \Lambda)$;
>   FUNCTION_TRACE3("%p␣=␣mp_const_dependency(%d)\n", $mp{\rightarrow}dep\_final$, $number\_to\_scaled(v)$);
>   **return** $mp{\rightarrow}dep\_final$;
> }

651.   And here's a more interesting way to start a dependency list from scratch: The parameter to *single_dependency* is the location of an independent variable x, and the result is the simple dependency list '$x + 0$'.

In the unlikely event that the given independent variable has been doubled so often that we can't refer to it with a nonzero coefficient, *single_dependency* returns the simple list '0'. This case can be recognized by testing that the returned list pointer is equal to *dep_final*.

#define   *two_to_the*(A)   (1 ≪ (**unsigned**)(A))

```
  static mp_value_node mp_single_dependency(MP mp, mp_node p)
  {
    mp_value_node q, rr;       /* the new dependency list */
    integer m;       /* the number of doublings */
    m = indep_scale(p);
    if (m > 28) {
      q = mp_const_dependency(mp, zero_t);
    }
    else {
      q = mp_get_dep_node(mp);
      set_dep_value(q, zero_t);
      set_number_from_scaled(dep_value(q), (integer) two_to_the(28 − m));
      set_dep_info(q, p);
      rr = mp_const_dependency(mp, zero_t);
      set_mp_link(q, (mp_node) rr);
    }
    FUNCTION_TRACE3("%p␣=␣mp_single_dependency(%p)\n", q, p);
    return q;
  }
```

652.   We sometimes need to make an exact copy of a dependency list.

```
  static mp_value_node mp_copy_dep_list(MP mp, mp_value_node p)
  {
    mp_value_node q;       /* the new dependency list */
    FUNCTION_TRACE2("mp_copy_dep_list(%p)\n", p);
    q = mp_get_dep_node(mp);
    mp→dep_final = q;
    while (1) {
      set_dep_info(mp→dep_final, dep_info(p));
      set_dep_value(mp→dep_final, dep_value(p));
      if (dep_info(mp→dep_final) ≡ Λ) break;
      set_mp_link(mp→dep_final, (mp_node) mp_get_dep_node(mp));
      mp→dep_final = (mp_value_node) mp_link(mp→dep_final);
      p = (mp_value_node) mp_link(p);
    }
    return q;
  }
```

653.   But how do variables normally become known? Ah, now we get to the heart of the equation-solving mechanism. The *linear_eq* procedure is given a *dependent* or *mp_proto_dependent* list, p, in which at least one independent variable appears. It equates this list to zero, by choosing an independent variable with the largest coefficient and making it dependent on the others. The newly dependent variable is eliminated from all current dependencies, thereby possibly making other dependent variables known.

The given list p is, of course, totally destroyed by all this processing.

static mp_value_node *find_node_with_largest_coefficient*(**MP** *mp*, **mp_value_node** *p*, **mp_number** ∗*v*);
static void *display_new_dependency*(**MP** *mp*, **mp_value_node** *p*, **mp_node** *x*, **integer** *n*);
static void *change_to_known*(**MP** *mp*, **mp_value_node** *p*, **mp_node** *x*, **mp_value_node** *final_node*, **integer** *n*);
static mp_value_node *divide_p_by_minusv_removing_q*(**MP** *mp*, **mp_value_node** *p*, **mp_value_node** *q*, **mp_value_node** ∗*final_node*, **mp_number** *v*, **quarterword** *t*);
static mp_value_node *divide_p_by_2_n*(**MP** *mp*, **mp_value_node** *p*, **integer** *n*);

static void *mp_linear_eq*(**MP** *mp*, **mp_value_node** *p*, **quarterword** *t*)
{
  mp_value_node *r*;   /∗ for link manipulation ∗/
  mp_node *x*;   /∗ the variable that loses its independence ∗/
  integer *n*;   /∗ the number of times *x* had been halved ∗/
  mp_number *v*;   /∗ the coefficient of *x* in list *p* ∗/
  mp_value_node *prev_r*;   /∗ lags one step behind *r* ∗/
  mp_value_node *final_node*;   /∗ the constant term of the new dependency list ∗/
  mp_value_node *qq*;

  new_number(*v*);
  FUNCTION_TRACE3("mp_linear_eq(%p,%d)\n", *p*, *t*);
  qq = *find_node_with_largest_coefficient*(*mp*, *p*, &*v*);
  x = *dep_info*(*qq*);
  n = *indep_scale*(*x*);
  p = *divide_p_by_minusv_removing_q*(*mp*, *p*, *qq*, &*final_node*, *v*, *t*);
  if (*number_positive*(*internal_value*(*mp_tracing_equations*))) {
    display_new_dependency(*mp*, *p*, (**mp_node**) *x*, *n*);
  }
  prev_r = (**mp_value_node**) *mp*→*dep_head*;
  r = (**mp_value_node**) *mp_link*(*mp*→*dep_head*);
  while (*r* ≠ *mp*→*dep_head*) {
    mp_value_node *s* = (**mp_value_node**) *dep_list*(*r*);
    mp_value_node *q* = *mp_p_with_x_becoming_q*(*mp*, *s*, *x*, (**mp_node**) *p*, *mp_type*(*r*));

    if (*dep_info*(*q*) ≡ Λ) {
      mp_make_known(*mp*, *r*, *q*);
    }
    else {
      set_dep_list(*r*, *q*);
      do {
        q = (**mp_value_node**) *mp_link*(*q*);
      } **while** (*dep_info*(*q*) ≠ Λ);
      prev_r = *q*;
    }
    r = (**mp_value_node**) *mp_link*(*prev_r*);
  }
  if (*n* > 0) {
    p = *divide_p_by_2_n*(*mp*, *p*, *n*);

```
    }
  change_to_known(mp, p, (mp_node) x, final_node, n);
  if (mp⃗fix_needed) mp_fix_dependencies(mp);
  free_number(v);
}
```

654.

```
static mp_value_node find_node_with_largest_coefficient(MP mp, mp_value_node p, mp_number *v)
{
  mp_number vabs;      /* its absolute value of v */
  mp_number rabs;      /* the absolute value of dep_value(r) */
  mp_value_node q = p;
  mp_value_node r = (mp_value_node) mp_link(p);
  new_number(vabs);
  new_number(rabs);
  number_clone(*v, dep_value(q));
  while (dep_info(r) ≠ Λ) {
    number_clone(vabs, *v);
    number_abs(vabs);
    number_clone(rabs, dep_value(r));
    number_abs(rabs);
    if (number_greater(rabs, vabs)) {
      q = r;
      number_clone(*v, dep_value(r));
    }
    r = (mp_value_node) mp_link(r);
  }
  free_number(vabs);
  free_number(rabs);
  return q;
}
```

655.   Here we want to change the coefficients from *scaled* to *fraction*, except in the constant term. In the common case of a trivial equation like '`x=3.14`', we will have $v = -fraction\_one$, $q = p$, and $t = mp\_dependent$.

```
static mp_value_node divide_p_by_minusv_removing_q(MP mp, mp_value_node p, mp_value_node
        q, mp_value_node *final_node, mp_number v, quarterword t)
{
    mp_value_node r;      /* for link manipulation */
    mp_value_node s;

    s = (mp_value_node) mp→temp_head;
    set_mp_link(s, (mp_node) p);
    r = p;
    do {
        if (r ≡ q) {
            set_mp_link(s, mp_link(r));
            mp_free_dep_node(mp, r);
        }
        else {
            mp_number w;      /* a tentative coefficient */
            mp_number absw;

            new_number(w);
            new_number(absw);
            make_fraction(w, dep_value(r), v);
            number_clone(absw, w);
            number_abs(absw);
            if (number_lessequal(absw, half_fraction_threshold_k)) {
                set_mp_link(s, mp_link(r));
                mp_free_dep_node(mp, r);
            }
            else {
                number_negate(w);
                set_dep_value(r, w);
                s = r;
            }
            free_number(w);
            free_number(absw);
        }
        r = (mp_value_node) mp_link(s);
    } while (dep_info(r) ≠ Λ);
    if (t ≡ mp_proto_dependent) {
        mp_number ret;

        new_number(ret);
        make_scaled(ret, dep_value(r), v);
        number_negate(ret);
        set_dep_value(r, ret);
        free_number(ret);
    }
    else if (number_to_scaled(v) ≠ −number_to_scaled(fraction_one_t)) {
        mp_number ret;

        new_fraction(ret);
        make_fraction(ret, dep_value(r), v);
        number_negate(ret);
```

```
      set_dep_value(r, ret);
      free_number(ret);
    }
    *final_node = r;
    return (mp_value_node) mp_link(mp→temp_head);
  }
```

656.

```
  static void display_new_dependency(MP mp, mp_value_node p, mp_node x, integer n)
  {
    if (mp_interesting(mp, x)) {
      int w0;

      mp_begin_diagnostic(mp);
      mp_print_nl(mp, "##␣");
      mp_print_variable_name(mp, x);
      w0 = n;
      while (w0 > 0) {
        mp_print(mp, "*4");
        w0 = w0 − 2;
      }
      mp_print_char(mp, xord('='));
      mp_print_dependency(mp, p, mp_dependent);
      mp_end_diagnostic(mp, false);
    }
  }
```

657.   The $n > 0$ test is repeated here because it is of vital importance to the function's functioning.

static mp_value_node *divide_p_by_2_n*(**MP** *mp*, **mp_value_node** *p*, **integer** *n*)
{
    mp_value_node *pp* = Λ;
    if $(n > 0)$ {   /* Divide list *p* by 2^n */
        mp_value_node *r*;
        mp_value_node *s*;
        mp_number *absw*;
        mp_number *w*;   /* a tentative coefficient */
        new_number(*w*);
        new_number(*absw*);
        s = (**mp_value_node**) *mp*→*temp_head*;
        set_mp_link(*mp*→*temp_head*, (**mp_node**) *p*);
        r = *p*;
        do {
            if $(n > 30)$ {
                set_number_to_zero(*w*);
            }
            else {
                number_clone(*w*, *dep_value*(*r*));
                number_divide_int(*w*, *two_to_the*(*n*));
            }
            number_clone(*absw*, *w*);
            number_abs(*absw*);
            if (*number_lessequal*(*absw*, *half_fraction_threshold_k*) ∧ (*dep_info*(*r*) ≠ Λ)) {
                set_mp_link(*s*, *mp_link*(*r*));
                mp_free_dep_node(*mp*, *r*);
            }
            else {
                set_dep_value(*r*, *w*);
                s = *r*;
            }
            r = (**mp_value_node**) *mp_link*(*s*);
        } **while** (*dep_info*(*s*) ≠ Λ);
        pp = (**mp_value_node**) *mp_link*(*mp*→*temp_head*);
        free_number(*absw*);
        free_number(*w*);
    }
    return *pp*;
}

658.

```
static void change_to_known(MP mp, mp_value_node p, mp_node x, mp_value_node
        final_node, integer n)
{
    if (dep_info(p) ≡ Λ) {
        mp_number absx;

        new_number(absx);
        mp_type(x) = mp_known;
        set_value_number(x, dep_value(p));
        number_clone(absx, value_number(x));
        number_abs(absx);
        if (number_greaterequal(absx, warning_limit_t))  mp_val_too_big(mp, value_number(x));
        free_number(absx);
        mp_free_dep_node(mp, p);
        if (cur_exp_node() ≡ x ∧ mp→cur_exp.type ≡ mp_independent) {
            set_cur_exp_value_number(value_number(x));
            mp→cur_exp.type = mp_known;
            mp_free_value_node(mp, x);
        }
    }
    else {
        mp→dep_final = final_node;
        mp_new_dep(mp, x, mp_dependent, p);
        if (cur_exp_node() ≡ x ∧ mp→cur_exp.type ≡ mp_independent) {
            mp→cur_exp.type = mp_dependent;
        }
    }
}
```

659.   Dynamic nonlinear equations.   Variables of numeric type are maintained by the general scheme of independent, dependent, and known values that we have just studied; and the components of pair and transform variables are handled in the same way. But METAPOST also has five other types of values: **boolean**, **string**, **pen**, **path**, and **picture**; what about them?

Equations are allowed between nonlinear quantities, but only in a simple form. Two variables that haven't yet been assigned values are either equal to each other, or they're not.

Before a boolean variable has received a value, its type is $mp\_unknown\_boolean$; similarly, there are variables whose type is $mp\_unknown\_string$, $mp\_unknown\_pen$, $mp\_unknown\_path$, and $mp\_unknown\_picture$. In such cases the value is either Λ (which means that no other variables are equivalent to this one), or it points to another variable of the same undefined type. The pointers in the latter case form a cycle of nodes, which we shall call a "ring." Rings of undefined variables may include capsules, which arise as intermediate results within expressions or as **expr** parameters to macros.

When one member of a ring receives a value, the same value is given to all the other members. In the case of paths and pictures, this implies making separate copies of a potentially large data structure; users should restrain their enthusiasm for such generality, unless they have lots and lots of memory space.

660.   The following procedure is called when a capsule node is being added to a ring (e.g., when an unknown variable is mentioned in an expression).

> **static mp_node** $mp\_new\_ring\_entry(\textbf{MP}\ mp, \textbf{mp\_node}\ p)$
> {
>   **mp_node** q;   /∗ the new capsule node ∗/
>
> $q = mp\_get\_value\_node(mp)$;
> $mp\_name\_type(q) = mp\_capsule$;
> $mp\_type(q) = mp\_type(p)$;
>   **if** $(value\_node(p) \equiv \Lambda)$ $set\_value\_node(q, p)$;
>   **else** $set\_value\_node(q, value\_node(p))$;
> $set\_value\_node(p, q)$;
>   **return** q;
> }

661.   Conversely, we might delete a capsule or a variable before it becomes known. The following procedure simply detaches a quantity from its ring, without recycling the storage.

⟨ Declarations 8 ⟩ +≡
> **static void** $mp\_ring\_delete(\textbf{MP}\ mp, \textbf{mp\_node}\ p)$;

662.   **void** $mp\_ring\_delete(\textbf{MP}\ mp, \textbf{mp\_node}\ p)$
> {
>   **mp_node** q;
>
>   (**void**) mp;
> $q = value\_node(p)$;
>   **if** $(q \neq \Lambda \wedge q \neq p)$ {
>   **while** $(value\_node(q) \neq p)$ $q = value\_node(q)$;
> $set\_value\_node(q, value\_node(p))$;
>   }
> }

663.   Eventually there might be an equation that assigns values to all of the variables in a ring. The *nonlinear_eq* subroutine does the necessary propagation of values.

  If the parameter *flush_p* is *true*, node p itself needn't receive a value, it will soon be recycled.

> **static void** *mp_nonlinear_eq*(**MP** *mp*, **mp_value** *v*, **mp_node** *p*, **boolean** *flush_p*)
> {
>   *mp_variable_type* *t*;   /∗ the type of ring p ∗/
>   **mp_node** *q*, *r*;   /∗ link manipulation registers ∗/
>   *t* = (*mp_type*(*p*) − *unknown_tag*);
>   *q* = *value_node*(*p*);
>   **if** (*flush_p*) *mp_type*(*p*) = *mp_vacuous*;
>   **else** *p* = *q*;
>   **do** {
>   *r* = *value_node*(*q*);
>   *mp_type*(*q*) = *t*;
>   **switch** (*t*) {
>   **case** *mp_boolean_type*: *set_value_number*(*q*, *v.data.n*);
>   **break**;
>   **case** *mp_string_type*: *set_value_str*(*q*, *v.data.str*);
>   *add_str_ref*(*v.data.str*);
>   **break**;
>   **case** *mp_pen_type*: *set_value_knot*(*q*, *copy_pen*(*v.data.p*));
>   **break**;
>   **case** *mp_path_type*: *set_value_knot*(*q*, *mp_copy_path*(*mp*, *v.data.p*));
>   **break**;
>   **case** *mp_picture_type*: *set_value_node*(*q*, *v.data.node*);
>   *add_edge_ref*(*v.data.node*);
>   **break**;
>   **default**: **break**;
>   }   /∗ there ain't no more cases ∗/
>   *q* = *r*;
>   } **while** ($q \neq p$);
> }

664.   If two members of rings are equated, and if they have the same type, the *ring_merge* procedure is called on to make them equivalent.

> **static void** *mp_ring_merge*(**MP** *mp*, **mp_node** *p*, **mp_node** *q*)
> {
>   **mp_node** *r*;   /∗ traverses one list ∗/
>   *r* = *value_node*(*p*);
>   **while** ($r \neq p$) {
>   **if** ($r \equiv q$) {
>   *exclaim_redundant_equation*(*mp*);
>   **return**;
>   }
>   ;
>   *r* = *value_node*(*r*);
>   }
>   *r* = *value_node*(*p*);
>   *set_value_node*(*p*, *value_node*(*q*));
>   *set_value_node*(*q*, *r*);
> }

665.   **static void** *exclaim_redundant_equation*(**MP** *mp*)
{
    const char *∗hlp*[ ] = {"I␣already␣knew␣that␣this␣equation␣was␣true.",
        "But␣perhaps␣no␣harm␣has␣been␣done;␣let's␣continue.", Λ};

    mp_back_error(*mp*, "Redundant␣equation", *hlp*, *true*);
    mp_get_x_next(*mp*);
}

666.   ⟨ Declarations 8 ⟩ +≡
  static void *exclaim_redundant_equation*(**MP** *mp*);

667.   Introduction to the syntactic routines.   Let's pause a moment now and try to look at the Big Picture. The METAPOST program consists of three main parts: syntactic routines, semantic routines, and output routines. The chief purpose of the syntactic routines is to deliver the user's input to the semantic routines, while parsing expressions and locating operators and operands. The semantic routines act as an interpreter responding to these operators, which may be regarded as commands. And the output routines are periodically called on to produce compact font descriptions that can be used for typesetting or for making interim proof drawings. We have discussed the basic data structures and many of the details of semantic operations, so we are good and ready to plunge into the part of METAPOST that actually controls the activities.

Our current goal is to come to grips with the *get_next* procedure, which is the keystone of METAPOST's input mechanism. Each call of *get_next* sets the value of three variables *cur_cmd*, *cur_mod*, and *cur_sym*, representing the next input token.

> *cur_cmd* denotes a command code from the long list of codes given earlier;
> *cur_mod* denotes a modifier or operand of the command code;
> *cur_sym* is the hash address of the symbolic token that was just scanned,
>   or zero in the case of a numeric or string or capsule token.

Underlying this external behavior of *get_next* is all the machinery necessary to convert from character files to tokens. At a given time we may be only partially finished with the reading of several files (for which **input** was specified), and partially finished with the expansion of some user-defined macros and/or some macro parameters, and partially finished reading some text that the user has inserted online, and so on. When reading a character file, the characters must be converted to tokens; comments and blank spaces must be removed, numeric and string tokens must be evaluated.

To handle these situations, which might all be present simultaneously, METAPOST uses various stacks that hold information about the incomplete activities, and there is a finite state control for each level of the input mechanism. These stacks record the current state of an implicitly recursive process, but the *get_next* procedure is not recursive.

#define *cur_cmd*() (**unsigned**)($mp\rightarrow cur\_mod\_\rightarrow type$)
#define *set_cur_cmd*(A) $mp\rightarrow cur\_mod\_\rightarrow type = (A)$
#define *cur_mod_int*()   *number_to_int*($mp\rightarrow cur\_mod\_\rightarrow data.n$)   /∗ operand of current command ∗/
#define *cur_mod*()   *number_to_scaled*($mp\rightarrow cur\_mod\_\rightarrow data.n$)   /∗ operand of current command ∗/
#define *cur_mod_number*() $mp\rightarrow cur\_mod\_\rightarrow data.n$ /∗ operand of current command ∗/
#define *set_cur_mod*(A)   *set_number_from_scaled*($mp\rightarrow cur\_mod\_\rightarrow data.n, (A)$)
#define *set_cur_mod_number*(A)   *number_clone*($mp\rightarrow cur\_mod\_\rightarrow data.n, (A)$)
#define *cur_mod_node*() $mp\rightarrow cur\_mod\_\rightarrow data.node$
#define *set_cur_mod_node*(A) $mp\rightarrow cur\_mod\_\rightarrow data.node = (A)$
#define *cur_mod_str*() $mp\rightarrow cur\_mod\_\rightarrow data.str$
#define *set_cur_mod_str*(A) $mp\rightarrow cur\_mod\_\rightarrow data.str = (A)$
#define *cur_sym*() $mp\rightarrow cur\_mod\_\rightarrow data.sym$
#define *set_cur_sym*(A) $mp\rightarrow cur\_mod\_\rightarrow data.sym = (A)$
#define *cur_sym_mod*() $mp\rightarrow cur\_mod\_\rightarrow name\_type$
#define *set_cur_sym_mod*(A) $mp\rightarrow cur\_mod\_\rightarrow name\_type = (A)$
⟨ Global variables 14 ⟩ +≡
  mp_node *cur_mod_*;   /∗ current command, symbol, and its operands ∗/

668.   ⟨ Initialize table entries 182 ⟩ +≡
  $mp\rightarrow cur\_mod\_ = mp\_get\_symbolic\_node(mp)$;

669.   ⟨ Free table entries 183 ⟩ +≡
  $mp\_free\_symbolic\_node(mp, mp\rightarrow cur\_mod\_)$;

670. The *print_cmd_mod* routine prints a symbolic interpretation of a command code and its modifier. It consists of a rather tedious sequence of print commands, and most of it is essentially an inverse to the *primitive* routine that enters a METAPOST primitive into *hash* and *eqtb*. Therefore almost all of this procedure appears elsewhere in the program, together with the corresponding *primitive* calls.

⟨ Declarations 8 ⟩ +≡
  static void *mp_print_cmd_mod*(**MP** *mp*, **integer** *c*, **integer** *m*);

671.　　**void** *mp_print_cmd_mod*(**MP** *mp*, **integer** *c*, **integer** *m*){ **switch** (*c*) { ⟨ Cases of *print_cmd_mod*
    for symbolic printing of primitives 233 ⟩
  default: *mp_print*(*mp*, "[unknown␣command␣code!]");
    break; } }

672. Here is a procedure that displays a given command in braces, in the user's transcript file.

#define *show_cur_cmd_mod*　　*mp_show_cmd_mod*(*mp*, *cur_cmd*( ), *cur_mod*( ))
  static void *mp_show_cmd_mod*(**MP** *mp*, **integer** *c*, **integer** *m*)
  {
    mp_begin_diagnostic(*mp*);
    mp_print_nl(*mp*, "{");
    mp_print_cmd_mod(*mp*, *c*, *m*);
    mp_print_char(*mp*, *xord*('}'));
    mp_end_diagnostic(*mp*, *false*);
  }

673.   Input stacks and states.   The state of METAPOST's input mechanism appears in the input stack, whose entries are records with five fields, called *index*, *start*, *loc*, *limit*, and *name*. The top element of this stack is maintained in a global variable for which no subscripting needs to be done; the other elements of the stack appear in an array. Hence the stack is declared thus:

⟨ Types in the outer block 33 ⟩ +≡
  typedef struct {
    char *∗long_name_field*;
    halfword *start_field*, *loc_field*, *limit_field*;
    mp_node *nstart_field*, *nloc_field*;
    mp_string *name_field*;
    quarterword *index_field*;
  } **in_state_record**;

674.   ⟨ Global variables 14 ⟩ +≡
  in_state_record *∗input_stack*;
  integer *input_ptr*;   /∗ first unused location of *input_stack* ∗/
  integer *max_in_stack*;   /∗ largest value of *input_ptr* when pushing ∗/
  in_state_record *cur_input*;   /∗ the "top" input state ∗/
  int *stack_size*;   /∗ maximum number of simultaneous input sources ∗/

675.   ⟨ Allocate or initialize variables 28 ⟩ +≡
  mp→stack_size = 16;
  mp→input_stack = *xmalloc*((*mp→stack_size* + 1), **sizeof**(**in_state_record**));

676.   ⟨ Dealloc variables 27 ⟩ +≡
  xfree(*mp→input_stack*);

677.   We've already defined the special variable *loc* ≡ *cur_input.loc_field* in our discussion of basic input-output routines. The other components of *cur_input* are defined in the same way:

#define *iindex* *mp→cur_input.index_field*   /∗ reference for buffer information ∗/
#define *start* *mp→cur_input.start_field*   /∗ starting position in *buffer* ∗/
#define *limit* *mp→cur_input.limit_field*   /∗ end of current line in *buffer* ∗/
#define *name* *mp→cur_input.name_field*   /∗ name of the current file ∗/

678.   Let's look more closely now at the five control variables (*index*, *start*, *loc*, *limit*, *name*), assuming that METAPOST is reading a line of characters that have been input from some file or from the user's terminal. There is an array called *buffer* that acts as a stack of all lines of characters that are currently being read from files, including all lines on subsidiary levels of the input stack that are not yet completed. METAPOST will return to the other lines when it is finished with the present input file.

(Incidentally, on a machine with byte-oriented addressing, it would be appropriate to combine *buffer* with the *str_pool* array, letting the buffer entries grow downward from the top of the string pool and checking that these two tables don't bump into each other.)

The line we are currently working on begins in position *start* of the buffer; the next character we are about to read is *buffer*[*loc*]; and *limit* is the location of the last character present. We always have *loc* ≤ *limit*. For convenience, *buffer*[*limit*] has been set to **"%"**, so that the end of a line is easily sensed.

The *name* variable is a string number that designates the name of the current file, if we are reading an ordinary text file. Special codes *is_term* .. *max_spec_src* indicate other sources of input text.

#define *is_term* (**mp_string**) 0   /∗ *name* value when reading from the terminal for normal input ∗/
#define *is_read* (**mp_string**) 1   /∗ *name* value when executing a **readstring** or **readfrom** ∗/
#define *is_scantok* (**mp_string**) 2   /∗ *name* value when reading text generated by **scantokens** ∗/
#define *max_spec_src* *is_scantok*

679.   Additional information about the current line is available via the *index* variable, which counts how many lines of characters are present in the buffer below the current level. We have *index* = 0 when reading from the terminal and prompting the user for each line; then if the user types, e.g., '`input figs`', we will have *index* = 1 while reading the file `figs.mp`. However, it does not follow that *index* is the same as the input stack pointer, since many of the levels on the input stack may come from token lists and some *index* values may correspond to `MPX` files that are not currently on the stack.

The global variable *in_open* is equal to the highest *index* value counting `MPX` files but excluding token-list input levels. Thus, the number of partially read lines in the buffer is *in_open* + 1 and we have *in_open* ≥ *index* when we are not reading a token list.

If we are not currently reading from the terminal, we are reading from the file variable *input_file*[*index*]. We use the notation *terminal_input* as a convenient abbreviation for *name* = *is_term*, and *cur_file* as an abbreviation for *input_file*[*index*].

When METAPOST is not reading from the terminal, the global variable **line** contains the line number in the current file, for use in error messages. More precisely, **line** is a macro for *line_stack*[*index*] and the *line_stack* array gives the line number for each file in the *input_file* array.

When an `MPX` file is opened the file name is stored in the *mpx_name* array so that the name doesn't get lost when the file is temporarily removed from the input stack. Thus when *input_file*[*k*] is an `MPX` file, its name is *mpx_name*[*k*] and it contains translated TEX pictures for *input_file*[*k* − 1]. Since this is not an `MPX` file, we have

$$mpx\_name[k-1] \leq absent.$$

This *name* field is set to *finished* when *input_file*[*k*] is completely read.

If more information about the input state is needed, it can be included in small arrays like those shown here. For example, the current page or segment number in the input file might be put into a variable *page*, that is really a macro for the current entry in '*page_stack*: *array*[0..*max_in_open*] *of* **integer**' by analogy with *line_stack*.

#define *terminal_input*   (*name* ≡ *is_term*)   /∗ are we reading from the terminal? ∗/
#define *cur_file*   *mp*→*input_file*[*iindex*]   /∗ the current **void** ∗ variable ∗/
#define **line**   *mp*→*line_stack*[*iindex*]   /∗ current line number in the current source file ∗/
#define *in_ext*   *mp*→*inext_stack*[*iindex*]   /∗ a string used to construct `MPX` file names ∗/
#define *in_name*   *mp*→*iname_stack*[*iindex*]   /∗ a string used to construct `MPX` file names ∗/
#define *in_area*   *mp*→*iarea_stack*[*iindex*]   /∗ another string for naming `MPX` files ∗/
#define *absent*   (**mp_string**) 1   /∗ *name_field* value for unused *mpx_in_stack* entries ∗/
#define *mpx_reading*   (*mp*→*mpx_name*[*iindex*] > *absent*)   /∗ when reading a file, is it an `MPX` file? ∗/
#define *mpx_finished*   0   /∗ *name_field* value when the corresponding `MPX` file is finished ∗/

⟨ Global variables 14 ⟩ +≡
  integer *in_open*;   /∗ the number of lines in the buffer, less one ∗/
  integer *in_open_max*;   /∗ highest value of *in_open* ever seen ∗/
  unsigned int *open_parens*;   /∗ the number of open text files ∗/
  void ∗∗*input_file*;
  integer ∗*line_stack*;   /∗ the line number for each file ∗/
  char ∗∗*inext_stack*;   /∗ used for naming `MPX` files ∗/
  char ∗∗*iname_stack*;   /∗ used for naming `MPX` files ∗/
  char ∗∗*iarea_stack*;   /∗ used for naming `MPX` files ∗/
  mp_string ∗*mpx_name*;

680.   ⟨ Declarations 8 ⟩ +≡
  static void *mp_reallocate_input_stack*(**MP** *mp*, **int** *newsize*);

681.   **static void** *mp_reallocate_input_stack* (**MP** *mp*, **int** *newsize*)
  {
    int *k*;
    int *n* = *newsize* + 1;

    XREALLOC(*mp*→*input_file*, *n*, **void** ∗);
    XREALLOC(*mp*→*line_stack*, *n*, **integer**);
    XREALLOC(*mp*→*inext_stack*, *n*, **char** ∗);
    XREALLOC(*mp*→*iname_stack*, *n*, **char** ∗);
    XREALLOC(*mp*→*iarea_stack*, *n*, **char** ∗);
    XREALLOC(*mp*→*mpx_name*, *n*, **mp_string**);
    for (*k* = *mp*→*max_in_open*; *k* ≤ *n*; *k*++) {
      mp→*input_file* [*k*] = Λ;
      mp→*line_stack* [*k*] = 0;
      mp→*inext_stack* [*k*] = Λ;
      mp→*iname_stack* [*k*] = Λ;
      mp→*iarea_stack* [*k*] = Λ;
      mp→*mpx_name* [*k*] = Λ;
    }
    mp→*max_in_open* = *newsize*;
  }

682.   This has to be more than *file_bottom*, so:
⟨ Allocate or initialize variables 28 ⟩ +≡
  mp_reallocate_input_stack (*mp*, *file_bottom* + 4);

683.   ⟨ Dealloc variables 27 ⟩ +≡
  {
    int *l*;

    for (*l* = 0; *l* ≤ *mp*→*max_in_open*; *l*++) {
      xfree (*mp*→*inext_stack* [*l*]);
      xfree (*mp*→*iname_stack* [*l*]);
      xfree (*mp*→*iarea_stack* [*l*]);
    }
  }
  xfree (*mp*→*input_file*);
  xfree (*mp*→*line_stack*);
  xfree (*mp*→*inext_stack*);
  xfree (*mp*→*iname_stack*);
  xfree (*mp*→*iarea_stack*);
  xfree (*mp*→*mpx_name*);

684.   However, all this discussion about input state really applies only to the case that we are inputting from a file. There is another important case, namely when we are currently getting input from a token list. In this case *iindex* > *max_in_open*, and the conventions about the other state variables are different:

nloc is a pointer to the current node in the token list, i.e., the node that will be read next. If *nloc* = Λ, the token list has been fully read.

start points to the first node of the token list; this node may or may not contain a reference count, depending on the type of token list involved.

token_type, which takes the place of *iindex* in the discussion above, is a code number that explains what kind of token list is being scanned.

name points to the *eqtb* address of the control sequence being expanded, if the current token list is a macro not defined by **vardef**. Macros defined by **vardef** have *name* = Λ; their name can be deduced by looking at their first two parameters.

param_start, which takes the place of *limit*, tells where the parameters of the current macro or loop text begin in the *param_stack*.

The *token_type* can take several values, depending on where the current token list came from:

    forever_text, if the token list being scanned is the body of a **forever** loop;
    loop_text, if the token list being scanned is the body of a **for** or **forsuffixes** loop;
    parameter, if a **text** or **suffix** parameter is being scanned;
    backed_up, if the token list being scanned has been inserted as 'to be read again'.
    inserted, if the token list being scanned has been inserted as part of error recovery;
    macro, if the expansion of a user-defined symbolic token is being scanned.

The token list begins with a reference count if and only if *token_type* = *macro*.

#define *nloc*   *mp→cur_input.nloc_field*   /∗ location of next node node ∗/
#define *nstart*   *mp→cur_input.nstart_field*   /∗ location of next node node ∗/
#define *token_type*   *iindex*   /∗ type of current token list ∗/
#define *token_state*   (*iindex* ≤ *macro*)   /∗ are we scanning a token list? ∗/
#define *file_state*   (*iindex* > *macro*)   /∗ are we scanning a file line? ∗/
#define *param_start*   *limit*   /∗ base of macro parameters in *param_stack* ∗/
#define *forever_text*   0   /∗ *token_type* code for loop texts ∗/
#define *loop_text*   1   /∗ *token_type* code for loop texts ∗/
#define *parameter*   2   /∗ *token_type* code for parameter texts ∗/
#define *backed_up*   3   /∗ *token_type* code for texts to be reread ∗/
#define *inserted*   4   /∗ *token_type* code for inserted texts ∗/
#define *macro*   5   /∗ *token_type* code for macro replacement texts ∗/
#define *file_bottom*   6   /∗ lowest file code ∗/

685.   The *param_stack* is an auxiliary array used to hold pointers to the token lists for parameters at the current level and subsidiary levels of input. This stack grows at a different rate from the others, and is dynamically reallocated when needed.

⟨ Global variables 14 ⟩ +≡
    mp_node *∗param_stack*;   /∗ token list pointers for parameters ∗/
    integer *param_ptr*;   /∗ first unused entry in *param_stack* ∗/
    integer *max_param_stack*;   /∗ largest value of *param_ptr* ∗/

686.   ⟨ Allocate or initialize variables 28 ⟩ +≡
    mp→param_stack = *xmalloc*((*mp→param_size* + 1), **sizeof**(**mp_node**));

687.   **static void** *mp_check_param_size*(**MP** *mp*, **int** *k*)
   {
      while ($k \geq mp\rightarrow param\_size$) {
         XREALLOC($mp\rightarrow param\_stack$, ($k + k/4$), **mp_node**);
         $mp\rightarrow param\_size = k + k/4$;
      }
   }

688.   ⟨Dealloc variables 27⟩ +≡
   xfree($mp\rightarrow param\_stack$);

689.   Notice that the **line** isn't valid when *token_state* is true because it depends on *iindex*. If we really need to know the line number for the topmost file in the iindex stack we use the following function. If a page number or other information is needed, this routine should be modified to compute it as well.

⟨Declarations 8⟩ +≡
   static integer *mp_true_line*(**MP** *mp*);

690.   **integer** *mp_true_line*(**MP** *mp*){ **int** k;   /∗ an index into the input stack ∗/
      if (*file_state* $\land$ (*name* > *max_spec_src*)) { **return line** ; }
      else {
         $k = mp\rightarrow input\_ptr$;
         while ((k > 0) $\land$ (($mp\rightarrow input\_stack$[($k - 1$)].*index_field* <
               file_bottom) $\lor$ ($mp\rightarrow input\_stack$[($k - 1$)].*name_field* $\leq$ *max_spec_src*)))
            {
            decr(k);
            }
         return (k > 0 ? $mp\rightarrow line\_stack$[($k - 1$) + *file_bottom*] : 0);
      }
      }

691.   Thus, the "current input state" can be very complicated indeed; there can be many levels and each level can arise in a variety of ways. The *show_context* procedure, which is used by METAPOST's error-reporting routine to print out the current input state on all levels down to the most recent line of characters from an input file, illustrates most of these conventions. The global variable *file_ptr* contains the lowest level that was displayed by this procedure.

⟨Global variables 14⟩ +≡
   integer *file_ptr*;   /∗ shallowest level shown by *show_context* ∗/

692.   The status at each level is indicated by printing two lines, where the first line indicates what was read so far and the second line shows what remains to be read. The context is cropped, if necessary, so that the first line contains at most *half_error_line* characters, and the second contains at most *error_line*. Non-current input levels whose *token_type* is '*backed_up*' are shown only if they have not been fully read.

```
void mp_show_context(MP mp)
{    /* prints where the scanner is */
  unsigned old_setting;        /* saved selector setting */

  ⟨Local variables for formatting calculations 698⟩;
  mp→file_ptr = mp→input_ptr;
  mp→input_stack[mp→file_ptr] = mp→cur_input;      /* store current state */
  while (1) {
    mp→cur_input = mp→input_stack[mp→file_ptr];      /* enter into the context */
    ⟨Display the current context 693⟩;
    if (file_state)
      if ((name > max_spec_src) ∨ (mp→file_ptr ≡ 0)) break;
    decr(mp→file_ptr);
  }
  mp→cur_input = mp→input_stack[mp→input_ptr];      /* restore original state */
}
```

693.   ⟨Display the current context 693⟩ ≡
```
if ((mp→file_ptr ≡ mp→input_ptr) ∨ file_state ∨ (token_type ≠ backed_up) ∨ (nloc ≠ Λ)) {
    /* we omit backed-up token lists that have already been read */
  mp→tally = 0;      /* get ready to count characters */
  old_setting = mp→selector;
  if (file_state) {
    ⟨Print location of current line 694⟩;
    ⟨Pseudoprint the line 701⟩;
  }
  else {
    ⟨Print type of token list 695⟩;
    ⟨Pseudoprint the token list 702⟩;
  }
  mp→selector = old_setting;      /* stop pseudoprinting */
  ⟨Print two lines using the tricky pseudoprinted information 700⟩;
}
```
This code is used in section 692.

694.   This routine should be changed, if necessary, to give the best possible indication of where the current line resides in the input file. For example, on some systems it is best to print both a page and line number.

⟨ Print location of current line 694 ⟩ ≡

  if (*name* > *max_spec_src*) {

    mp_print_nl(*mp*, "l.");

    mp_print_int(*mp*, *mp_true_line*(*mp*));

  }

  else if (*terminal_input*) {

    if (*mp*⁻*file_ptr* ≡ 0) *mp_print_nl*(*mp*, "<*>");

    else *mp_print_nl*(*mp*, "<insert>");

  }

  else if (*name* ≡ *is_scantok*) {

    mp_print_nl(*mp*, "<scantokens>");

  }

  else {

    mp_print_nl(*mp*, "<read>");

  }

  mp_print_char(*mp*, *xord*('␣'))

This code is used in section 693.

695.   Can't use case statement here because the *token_type* is not a constant expression.

⟨ Print type of token list 695 ⟩ ≡

  {

    if (*token_type* ≡ *forever_text*) {

      mp_print_nl(*mp*, "<forever>␣");

    }

    else if (*token_type* ≡ *loop_text*) {

      ⟨ Print the current loop value 696 ⟩;

    }

    else if (*token_type* ≡ *parameter*) {

      mp_print_nl(*mp*, "<argument>␣");

    }

    else if (*token_type* ≡ *backed_up*) {

      if (*nloc* ≡ Λ) *mp_print_nl*(*mp*, "<recently␣read>␣");

      else *mp_print_nl*(*mp*, "<to␣be␣read␣again>␣");

    }

    else if (*token_type* ≡ *inserted*) {

      mp_print_nl(*mp*, "<inserted␣text>␣");

    }

    else if (*token_type* ≡ *macro*) {

      mp_print_ln(*mp*);

      if (*name* ≠ Λ) *mp_print_str*(*mp*, *name*);

      else ⟨ Print the name of a **vardef**'d macro 697 ⟩;

      mp_print(*mp*, "->");

    }

    else {

      mp_print_nl(*mp*, "?");   /∗ this should never happen ∗/

    }

  }

This code is used in section 693.

696.   The parameter that corresponds to a loop text is either a token list (in the case of **forsuffixes**) or a "capsule" (in the case of **for**). We'll discuss capsules later; for now, all we need to know is that the *link* field in a capsule parameter is **void** and that $print\_exp(p, 0)$ displays the value of capsule p in abbreviated form.

⟨ Print the current loop value 696 ⟩ ≡
```
  {
    mp_node pp;
    mp_print_nl (mp, "<for(");
    pp = mp→param_stack [param_start ];
    if (pp ≠ Λ) {
      if (mp_link (pp) ≡ MP_VOID)  mp_print_exp (mp, pp, 0);      /∗ we're in a for loop ∗/
      else  mp_show_token_list (mp, pp, Λ, 20, mp→tally );
    }
    mp_print (mp, ")>␣");
  }
```
This code is used in section 695.

697.   The first two parameters of a macro defined by **vardef** will be token lists representing the macro's prefix and "at point." By putting these together, we get the macro's full name.

⟨ Print the name of a **vardef**'d macro 697 ⟩ ≡
```
  {
    mp_node pp = mp→param_stack [param_start ];
    if (pp ≡ Λ) {
      mp_show_token_list (mp, mp→param_stack [param_start + 1], Λ, 20, mp→tally );
    }
    else {
      mp_node qq = pp;
      while (mp_link (qq) ≠ Λ)  qq = mp_link (qq);
      mp_link (qq) = mp→param_stack [param_start + 1];
      mp_show_token_list (mp, pp, Λ, 20, mp→tally );
      mp_link (qq) = Λ;
    }
  }
```
This code is used in section 695.

698.   Now it is necessary to explain a little trick. We don't want to store a long string that corresponds to a token list, because that string might take up lots of memory; and we are printing during a time when an error message is being given, so we dare not do anything that might overflow one of METAPOST's tables. So 'pseudoprinting' is the answer: We enter a mode of printing that stores characters into a buffer of length $error\_line$, where character $k + 1$ is placed into $trick\_buf[k \bmod error\_line]$ if $k < trick\_count$, otherwise character k is dropped. Initially we set $tally := 0$ and $trick\_count := 1000000$; then when we reach the point where transition from line 1 to line 2 should occur, we set $first\_count := tally$ and $trick\_count := \max(error\_line, tally + 1 + error\_line - half\_error\_line)$. At the end of the pseudoprinting, the values of $first\_count$, $tally$, and $trick\_count$ give us all the information we need to print the two lines, and all of the necessary text is in $trick\_buf$.

Namely, let l be the length of the descriptive information that appears on the first line. The length of the context information gathered for that line is $k = first\_count$, and the length of the context information gathered for line 2 is $m = \min(tally, trick\_count) - k$. If $l + k \leq h$, where $h = half\_error\_line$, we print $trick\_buf[0..k-1]$ after the descriptive information on line 1, and set $n := l + k$; here n is the length of line 1. If $l + k > h$, some cropping is necessary, so we set $n := h$ and print '...' followed by

$$trick\_buf[(l + k - h + 3)\ ..\ k - 1],$$

where subscripts of $trick\_buf$ are circular modulo $error\_line$. The second line consists of n spaces followed by $trick\_buf[k\ ..\ (k + m - 1)]$, unless $n + m > error\_line$; in the latter case, further cropping is done. This is easier to program than to explain.

⟨ Local variables for formatting calculations 698 ⟩ ≡
    int i;   /∗ index into $buffer$ ∗/
    integer l;   /∗ length of descriptive information on line 1 ∗/
    integer m;   /∗ context information gathered for line 2 ∗/
    int n;   /∗ length of line 1 ∗/
    integer p;   /∗ starting or ending place in $trick\_buf$ ∗/
    integer q;   /∗ temporary index ∗/
This code is used in section 692.

699.   The following code tells the print routines to gather the desired information.

#define $begin\_pseudoprint$
    {
      $l = mp\text{→}tally$;
      $mp\text{→}tally = 0$;
      $mp\text{→}selector = pseudo$;
      $mp\text{→}trick\_count = 1000000$;
    }
#define $set\_trick\_count()$
    {
      $mp\text{→}first\_count = mp\text{→}tally$;
      $mp\text{→}trick\_count = mp\text{→}tally + 1 + mp\text{→}error\_line - mp\text{→}half\_error\_line$;
      if $(mp\text{→}trick\_count < mp\text{→}error\_line)$ $mp\text{→}trick\_count = mp\text{→}error\_line$;
    }

700.   And the following code uses the information after it has been gathered.

⟨ Print two lines using the tricky pseudoprinted information 700 ⟩ ≡
    if $(mp$-$trick\_count \equiv 1000000)$ $set\_trick\_count()$;   /* set_trick_count must be performed */
    if $(mp$-$tally < mp$-$trick\_count)$ $m = mp$-$tally - mp$-$first\_count$;
    else $m = mp$-$trick\_count - mp$-$first\_count$;   /* context on line 2 */
    if $(l + mp$-$first\_count \leq mp$-$half\_error\_line)$ {
      $p = 0$;
      $n = l + mp$-$first\_count$;
    }
    else {
      $mp\_print(mp, "...")$;
      $p = l + mp$-$first\_count - mp$-$half\_error\_line + 3$;
      $n = mp$-$half\_error\_line$;
    }
    for $(q = p;\ q \leq mp$-$first\_count - 1;\ q{+}{+})$ {
      $mp\_print\_char(mp, mp$-$trick\_buf[q \% mp$-$error\_line])$;
    }
    $mp\_print\_ln(mp)$;
    for $(q = 1;\ q \leq n;\ q{+}{+})$ {
      $mp\_print\_char(mp, xord('\_'))$;   /* print n spaces to begin line 2 */
    }
    if $(m + n \leq mp$-$error\_line)$ $p = mp$-$first\_count + m$;
    else $p = mp$-$first\_count + (mp$-$error\_line - n - 3)$;
    for $(q = mp$-$first\_count;\ q \leq p - 1;\ q{+}{+})$ {
      $mp\_print\_char(mp, mp$-$trick\_buf[q \% mp$-$error\_line])$;
    }
    if $(m + n > mp$-$error\_line)$ $mp\_print(mp, "...")$
This code is used in section 693.

701.   But the trick is distracting us from our current goal, which is to understand the input state. So let's concentrate on the data structures that are being pseudoprinted as we finish up the *show_context* procedure.

⟨ Pseudoprint the line 701 ⟩ ≡
  $begin\_pseudoprint$;
  if $(limit > 0)$ {
    for $(i = start;\ i \leq limit - 1;\ i{+}{+})$ {
      if $(i \equiv loc)$ $set\_trick\_count()$;
      $mp\_print\_char(mp, mp$-$buffer[i])$;
    }
  }
This code is used in section 693.

702.   ⟨ Pseudoprint the token list 702 ⟩ ≡
  $begin\_pseudoprint$;
  if $(token\_type \neq macro)$ $mp\_show\_token\_list(mp, nstart, nloc, 100000, 0)$;
  else $mp\_show\_macro(mp, nstart, nloc, 100000)$
This code is used in section 693.

703.   Maintaining the input stacks.   The following subroutines change the input status in commonly needed ways.

First comes *push_input*, which stores the current state and creates a new level (having, initially, the same properties as the old).

#define *push_input*
```
        {      /∗ enter a new input level, save the old ∗/
          if (mp→input_ptr > mp→max_in_stack) {
            mp→max_in_stack = mp→input_ptr;
            if (mp→input_ptr ≡ mp→stack_size) {
              int l = (mp→stack_size + (mp→stack_size/4));
              XREALLOC(mp→input_stack, l, in_state_record);
              mp→stack_size = l;
            }
          }
          mp→input_stack[mp→input_ptr] = mp→cur_input;      /∗ stack the record ∗/
          incr(mp→input_ptr);
        }
```

704.   And of course what goes up must come down.

#define *pop_input*
```
        {      /∗ leave an input level, re-enter the old ∗/
          decr(mp→input_ptr);
          mp→cur_input = mp→input_stack[mp→input_ptr];
        }
```

705.   Here is a procedure that starts a new level of token-list input, given a token list p and its type t. If $t = macro$, the calling routine should set *name*, reset *loc*, and increase the macro's reference count.

#define *back_list*(A)   *mp_begin_token_list*(mp, (A), (**quarterword**) *backed_up*)
```
          /∗ backs up a simple token list ∗/
```
```
  static void mp_begin_token_list(MP mp, mp_node p, quarterword t)
  {
    push_input;
    nstart = p;
    token_type = t;
    param_start = mp→param_ptr;
    nloc = p;
  }
```

706.   When a token list has been fully scanned, the following computations should be done as we leave
that level of input.

```
static void mp_end_token_list(MP mp)
{     /* leave a token-list input level */
  mp_node p;     /* temporary register */
  if (token_type ≥ backed_up) {     /* token list to be deleted */
    if (token_type ≤ inserted) {
      mp_flush_token_list(mp, nstart);
      goto DONE;
    }
    else {
      mp_delete_mac_ref(mp, nstart);     /* update reference count */
    }
  }
  while (mp→param_ptr > param_start) {     /* parameters must be flushed */
    decr(mp→param_ptr);
    p = mp→param_stack[mp→param_ptr];
    if (p ≠ Λ) {
      if (mp_link(p) ≡ MP_VOID) {     /* it's an expr parameter */
        mp_recycle_value(mp, p);
        mp_free_value_node(mp, p);
      }
      else {
        mp_flush_token_list(mp, p);     /* it's a suffix or text parameter */
      }
    }
  }
DONE: pop_input;
  check_interrupt;
}
```

707.   The contents of *cur_cmd*, *cur_mod*, *cur_sym* are placed into an equivalent token by the *cur_tok* routine.

⟨ Declare the procedure called *make_exp_copy* 940 ⟩;

static mp_node *mp_cur_tok*(**MP** *mp*)
{
   mp_node *p*;   /∗ a new token node ∗/
  if (*cur_sym*( ) ≡ Λ ∧ *cur_sym_mod*( ) ≡ 0) {
    if (*cur_cmd*( ) ≡ *mp_capsule_token*) {
      mp_number *save_exp_num*;   /∗ possible *cur_exp* numerical to be restored ∗/
      mp_value *save_exp* = *mp*→*cur_exp*;   /∗ *cur_exp* to be restored ∗/
      new_number(*save_exp_num*);
      number_clone(*save_exp_num*, *cur_exp_value_number*( ));
      mp_make_exp_copy(*mp*, *cur_mod_node*( ));
      p = *mp_stash_cur_exp*(*mp*);
      mp_link(*p*) = Λ;
      mp→*cur_exp* = *save_exp*;
      number_clone(*mp*→*cur_exp.data.n*, *save_exp_num*);
      free_number(*save_exp_num*);
    }
    else {
      p = *mp_get_token_node*(*mp*);
      mp_name_type(*p*) = *mp_token*;
      if (*cur_cmd*( ) ≡ *mp_numeric_token*) {
        set_value_number(*p*, *cur_mod_number*( ));
        mp_type(*p*) = *mp_known*;
      }
      else {
        set_value_str(*p*, *cur_mod_str*( ));
        mp_type(*p*) = *mp_string_type*;
      }
    }
  }
  else {
    p = *mp_get_symbolic_node*(*mp*);
    set_mp_sym_sym(*p*, *cur_sym*( ));
    mp_name_type(*p*) = *cur_sym_mod*( );
  }
  return *p*;
}

708.   Sometimes METAPOST has read too far and wants to "unscan" what it has seen. The *back_input* procedure takes care of this by putting the token just scanned back into the input stream, ready to be read again. If *cur_sym* <> 0, the values of *cur_cmd* and *cur_mod* are irrelevant.

⟨ Declarations 8 ⟩ +≡
  static void *mp_back_input*(**MP** *mp*);

709.   **void** $mp\_back\_input($**MP** $mp)$
$\{$ /∗ undoes one token of input ∗/
  mp_node p;   /∗ a token list of length one ∗/
  $p = mp\_cur\_tok(mp)$;
  while $(token\_state \wedge (nloc \equiv \Lambda))$ $mp\_end\_token\_list(mp)$;   /∗ conserve stack space ∗/
  $back\_list(p)$;
$\}$

710.   The $back\_error$ routine is used when we want to restore or replace an offending token just before issuing an error message. We disable interrupts during the call of $back\_input$ so that the help message won't be lost.

⟨ Declarations 8 ⟩ +≡
  static void $mp\_back\_error($**MP** $mp,$ **const char** $∗msg,$ **const char** $∗∗hlp,$ **boolean** $deletions\_allowed)$;

711.   **static void** $mp\_back\_error($**MP** $mp,$ **const char** $∗msg,$ **const char** $∗∗hlp,$ **boolean**
          $deletions\_allowed)$
$\{$ /∗ back up one token and call **error** ∗/
  $mp{\rightarrow}OK\_to\_interrupt = false$;
  $mp\_back\_input(mp)$;
  $mp{\rightarrow}OK\_to\_interrupt = true$;
  $mp\_error(mp, msg, hlp, deletions\_allowed)$;
$\}$
  static void $mp\_ins\_error($**MP** $mp,$ **const char** $∗msg,$ **const char** $∗∗hlp,$ **boolean** $deletions\_allowed)$
$\{$ /∗ back up one inserted token and call **error** ∗/
  $mp{\rightarrow}OK\_to\_interrupt = false$;
  $mp\_back\_input(mp)$;
  $token\_type = ($**quarterword**$)$ $inserted$;
  $mp{\rightarrow}OK\_to\_interrupt = true$;
  $mp\_error(mp, msg, hlp, deletions\_allowed)$;
$\}$

712.   The $begin\_file\_reading$ procedure starts a new level of input for lines of characters to be read from a file, or as an insertion from the terminal. It does not take care of opening the file, nor does it set loc or $limit$ or **line**.

  void $mp\_begin\_file\_reading($**MP** $mp)$
  $\{$
    if $(mp{\rightarrow}in\_open \equiv (mp{\rightarrow}max\_in\_open - 1))$
      $mp\_reallocate\_input\_stack(mp, (mp{\rightarrow}max\_in\_open + mp{\rightarrow}max\_in\_open/4))$;
    if $(mp{\rightarrow}first \equiv mp{\rightarrow}buf\_size)$ $mp\_reallocate\_buffer(mp, (mp{\rightarrow}buf\_size + mp{\rightarrow}buf\_size/4))$;
    $mp{\rightarrow}in\_open{+}{+}$;
    $push\_input$;
    $iindex = ($**quarterword**$)$ $mp{\rightarrow}in\_open$;
    if $(mp{\rightarrow}in\_open\_max < mp{\rightarrow}in\_open)$ $mp{\rightarrow}in\_open\_max = mp{\rightarrow}in\_open$;
    $mp{\rightarrow}mpx\_name[iindex] = absent$;
    $start = ($**halfword**$)$ $mp{\rightarrow}first$;
    $name = is\_term$;   /∗ $terminal\_input$ is now $true$ ∗/
  $\}$

713.   Conversely, the variables must be downdated when such a level of input is finished. Any associated MPX file must also be closed and popped off the file stack. While finishing preloading, it is possible that the file does not actually end with 'dump', so we capture that case here as well.

```
static void mp_end_file_reading(MP mp)
{
  if (mp→reading_preload ∧ mp→input_ptr ≡ 0) {
    set_cur_sym(mp→frozen_dump);
    mp_back_input(mp);
    return;
  }
  if (mp→in_open > iindex) {
    if ((mp→mpx_name[mp→in_open] ≡ absent) ∨ (name ≤ max_spec_src)) {
      mp_confusion(mp, "endinput");
      ;
    }
    else {
      (mp→close_file)(mp, mp→input_file[mp→in_open]);      /* close an MPX file */
      delete_str_ref(mp→mpx_name[mp→in_open]);
      decr(mp→in_open);
    }
  }
  mp→first = (size_t) start;
  if (iindex ≠ mp→in_open)  mp_confusion(mp, "endinput");
  if (name > max_spec_src) {
    (mp→close_file)(mp, cur_file);
    xfree(in_ext);
    xfree(in_name);
    xfree(in_area);
  }
  pop_input;
  decr(mp→in_open);
}
```

714.   Here is a function that tries to resume input from an `MPX` file already associated with the current input file. It returns *false* if this doesn't work.

> **static boolean** $mp\_begin\_mpx\_reading$(**MP** mp)
> {
>   **if** ($mp{\rightarrow}in\_open \neq iindex + 1$) {
>   **return** $false$;
>   }
>   **else** {
>   **if** ($mp{\rightarrow}mpx\_name[mp{\rightarrow}in\_open] \leq absent$) $mp\_confusion(mp, \texttt{"mpx"})$;
>   **if** ($mp{\rightarrow}first \equiv mp{\rightarrow}buf\_size$) $mp\_reallocate\_buffer(mp, (mp{\rightarrow}buf\_size + (mp{\rightarrow}buf\_size/4)))$;
> $push\_input$;
> $iindex = (\textbf{quarterword})\ mp{\rightarrow}in\_open$;
> $start = (\textbf{halfword})\ mp{\rightarrow}first$;
> $name = mp{\rightarrow}mpx\_name[mp{\rightarrow}in\_open]$;
> $add\_str\_ref(name)$;   /∗ Put an empty line in the input buffer ∗/
>   /∗ We want to make it look as though we have just read a blank line without really doing so. ∗/
> $mp{\rightarrow}last = mp{\rightarrow}first$;
> $limit = (\textbf{halfword})\ mp{\rightarrow}last$;   /∗ simulate *input_ln* and *firm_up_the_line* ∗/
> $mp{\rightarrow}buffer[limit] = xord(\text{'}\%\text{'})$;
> $mp{\rightarrow}first = (\textbf{size\_t})(limit + 1)$;
> $loc = start$;
>   **return** $true$;
>   }
> }

715.   This procedure temporarily stops reading an `MPX` file.

> **static void** $mp\_end\_mpx\_reading$(**MP** mp)
> {
>   **if** ($mp{\rightarrow}in\_open \neq iindex$) $mp\_confusion(mp, \texttt{"mpx"})$;
>   ;
>   **if** ($loc < limit$) {   /∗ Complain that we are not at the end of a line in the `MPX` file ∗/
>   /∗ Here we enforce a restriction that simplifies the input stacks considerably. This should not
>   inconvenience the user because `MPX` files are generated by an auxiliary program called `DVItoMP`.
>   ∗/
>   **const char** $*hlp[\,] = \{\texttt{"This\_file\_contains\_picture\_expressions\_for\_btex...etex"},$
> $\texttt{"blocks.\_\_Such\_files\_are\_normally\_generated\_automatically"},$
> $\texttt{"but\_this\_one\_seems\_to\_be\_messed\_up.\_\_I'm\_going\_to\_ignore"},$
> $\texttt{"the\_rest\_of\_this\_line."}, \Lambda\}$;
>
> $mp\_error(mp, \texttt{"'mpxbreak'\_must\_be\_at\_the\_end\_of\_a\_line"}, hlp, true)$;
>   }
> $mp{\rightarrow}first = (\textbf{size\_t})\ start$;
> pop_input;
> }

716.   In order to keep the stack from overflowing during a long sequence of inserted '`show`' commands, the following routine removes completed error-inserted lines from memory.

>   **void** *mp_clear_for_error_prompt*(**MP** *mp*)
>   {
>   **while** (*file_state* $\wedge$ *terminal_input* $\wedge$ (*mp*→*input_ptr* > 0) $\wedge$ (*loc* $\equiv$ *limit*)) *mp_end_file_reading*(*mp*);
>   *mp_print_ln*(*mp*);
>   *clear_terminal*( );
>   }

717.   To get METAPOST's whole input mechanism going, we perform the following actions.

⟨ Initialize the input routines 717 ⟩ ≡
>   { *mp*→*input_ptr* = 0;
>   *mp*→*max_in_stack* = *file_bottom*;
>   *mp*→*in_open* = *file_bottom*;
>   *mp*→*open_parens* = 0;
>   *mp*→*max_buf_stack* = 0;
>   *mp*→*param_ptr* = 0;
>   *mp*→*max_param_stack* = 0;
>   *mp*→*first* = 0;
>   *start* = 0;
>   *iindex* = *file_bottom*; **line** = 0;
>   *name* = *is_term*;
>   *mp*→*mpx_name*[*file_bottom*] = *absent*;
>   *mp*→*force_eof* = *false*;
>   **if** (¬*mp_init_terminal*(*mp*)) *mp_jump_out*(*mp*);
>   *limit* = (**halfword**) *mp*→*last*;
>   *mp*→*first* = *mp*→*last* + 1;   /∗ *init_terminal* has set *loc* and *last* ∗/
>   }

See also section 720.

This code is used in section 1298.

718.   Getting the next token.   The heart of METAPOST's input mechanism is the *get_next* procedure, which we shall develop in the next few sections of the program. Perhaps we shouldn't actually call it the "heart," however; it really acts as METAPOST's eyes and mouth, reading the source files and gobbling them up. And it also helps METAPOST to regurgitate stored token lists that are to be processed again.

The main duty of *get_next* is to input one token and to set *cur_cmd* and *cur_mod* to that token's command code and modifier. Furthermore, if the input token is a symbolic token, that token's *hash* address is stored in *cur_sym*; otherwise *cur_sym* is set to zero.

Underlying this simple description is a certain amount of complexity because of all the cases that need to be handled. However, the inner loop of *get_next* is reasonably short and fast.

719.   Before getting into *get_next*, we need to consider a mechanism by which METAPOST helps keep errors from propagating too far. Whenever the program goes into a mode where it keeps calling *get_next* repeatedly until a certain condition is met, it sets *scanner_status* to some value other than *normal*. Then if an input file ends, or if an '**outer**' symbol appears, an appropriate error recovery will be possible.

The global variable *warning_info* helps in this error recovery by providing additional information. For example, *warning_info* might indicate the name of a macro whose replacement text is being scanned.

#**define** *normal* 0   /∗ *scanner_status* at "quiet times" ∗/
#**define** *skipping* 1   /∗ *scanner_status* when false conditional text is being skipped ∗/
#**define** *flushing* 2   /∗ *scanner_status* when junk after a statement is being ignored ∗/
#**define** *absorbing* 3   /∗ *scanner_status* when a **text** parameter is being scanned ∗/
#**define** *var_defining* 4   /∗ *scanner_status* when a **vardef** is being scanned ∗/
#**define** *op_defining* 5   /∗ *scanner_status* when a macro **def** is being scanned ∗/
#**define** *loop_defining* 6   /∗ *scanner_status* when a **for** loop is being scanned ∗/
⟨ Global variables 14 ⟩ +≡
#**define** *tex_flushing* 7   /∗ *scanner_status* when skipping TEX material ∗/
  integer *scanner_status*;   /∗ are we scanning at high speed? ∗/
  mp_sym *warning_info*;   /∗ if so, what else do we need to know, in case an error occurs? ∗/
  integer *warning_line*;
  mp_node *warning_info_node*;

720.   ⟨ Initialize the input routines 717 ⟩ +≡
  mp→scanner_status = normal;

721.   The following subroutine is called when an '**outer**' symbolic token has been scanned or when the end of a file has been reached. These two cases are distinguished by *cur_sym*, which is zero at the end of a file.

> **static boolean** *mp_check_outer_validity* (**MP** *mp*)
> {
>   **mp_node** *p*;   /* points to inserted token list */
>   **if** (*mp*→*scanner_status* ≡ *normal*) {
>   **return** *true*;
>   }
>   **else if** (*mp*→*scanner_status* ≡ *tex_flushing*) {
>   ⟨Check if the file has ended while flushing TEX material and set the result value for
>   *check_outer_validity* 722⟩;
>   }
>   **else** {
>   ⟨Back up an outer symbolic token so that it can be reread 723⟩;
>   **if** (*mp*→*scanner_status* > *skipping*) {
>   ⟨Tell the user what has run away and try to recover 724⟩;
>   }
>   **else** {
>   **char** *msg*[256];
>   **const char** **hlp*[] = {"A␣forbidden␣'outer'␣token␣occurred␣in␣skipped␣text.",
>   "This␣kind␣of␣error␣happens␣when␣you␣say␣'if...'␣and␣forget",
>   "the␣matching␣'fi'.␣I've␣inserted␣a␣'fi';␣this␣might␣work.", Λ};
>
>   *mp_snprintf* (*msg*, 256, "Incomplete␣if;␣all␣text␣was␣ignored␣after␣line␣%d", (**int**)
>   *mp*→*warning_line*);
>   ;
>   **if** (*cur_sym* ( ) ≡ Λ) {
>   *hlp*[0] = "The␣file␣ended␣while␣I␣was␣skipping␣conditional␣text.";
>   }
>   *set_cur_sym* (*mp*→*frozen_fi*);
>   *mp_ins_error* (*mp*, *msg*, *hlp*, *false*);
>   }
>   **return** *false*;
>   }
> }

722.   ⟨Check if the file has ended while flushing TEX material and set the result value for
    check_outer_validity 722⟩ ≡
  if (*cur_sym*( ) ≠ Λ) {
    return *true*;
  }
  else {
    char *msg*[256];
    const char *∗hlp*[ ] = {"The␣file␣ended␣while␣I␣was␣looking␣for␣the␣'etex'␣to",
        "finish␣this␣TeX␣material.␣␣I've␣inserted␣'etex'␣now.", Λ};
    mp_snprintf (*msg*, 256, "TeX␣mode␣didn't␣end;␣all␣text␣was␣ignored␣after␣line␣%d", (**int**)
        mp→*warning_line*);
    set_cur_sym (*mp*→*frozen_etex*);
    mp_ins_error (*mp*, *msg*, *hlp*, *false*);
    return *false*;
  }

This code is used in section 721.

723.   ⟨Back up an outer symbolic token so that it can be reread 723⟩ ≡
  if (*cur_sym*( ) ≠ Λ) {
    p = *mp_get_symbolic_node* (*mp*);
    set_mp_sym_sym (*p*, *cur_sym*( ));
    mp_name_type (*p*) = *cur_sym_mod* ( );
    back_list (*p*);   /∗ prepare to read the symbolic token again ∗/
  }

This code is used in section 721.

724.   ⟨Tell the user what has run away and try to recover 724⟩ ≡
  {
    char *msg*[256];
    const char *∗msg_start* = Λ;
    const char *∗hlp*[ ] = {"I␣suspect␣you␣have␣forgotten␣an␣'enddef',",
        "causing␣me␣to␣read␣past␣where␣you␣wanted␣me␣to␣stop.",
        "I'll␣try␣to␣recover;␣but␣if␣the␣error␣is␣serious,",
        "you'd␣better␣type␣'E'␣or␣'X'␣now␣and␣fix␣your␣file.", Λ};
    mp_runaway (*mp*);   /∗ print the definition-so-far ∗/
    if (*cur_sym*( ) ≡ Λ) {
      msg_start = "File␣ended␣while␣scanning";
    }
    else {
      msg_start = "Forbidden␣token␣found␣while␣scanning";
    }
    switch (*mp*→*scanner_status*) {⟨Complete the error message, and set *cur_sym* to a token that might
        help recover from the error 725⟩}   /∗ there are no other cases ∗/
    mp_ins_error (*mp*, *msg*, *hlp*, *true*);
  }

This code is used in section 721.

725.   As we consider various kinds of errors, it is also appropriate to change the first line of the help message just given; *help_line*[3] points to the string that might be changed.

⟨ Complete the error message, and set *cur_sym* to a token that might help recover from the error 725 ⟩ ≡
case *flushing*: *mp_snprintf* (*msg*, 256, "%s␣to␣the␣end␣of␣the␣statement", *msg_start*);
  hlp[0] = "A␣previous␣error␣seems␣to␣have␣propagated,";
  set_cur_sym(*mp→frozen_semicolon*);
  break;
case *absorbing*: *mp_snprintf* (*msg*, 256, "%s␣a␣text␣argument", *msg_start*);
  hlp[0] = "It␣seems␣that␣a␣right␣delimiter␣was␣left␣out,";
  if (*mp→warning_info* ≡ Λ) {
    set_cur_sym(*mp→frozen_end_group*);
  }
  else {
    set_cur_sym(*mp→frozen_right_delimiter*);   /* the next line makes sure that the inserted delimiter
        will match the delimiter that already was read. */
    set_equiv_sym(*cur_sym*( ), *mp→warning_info*);
  }
  break;
case *var_defining*:
  {
    mp_string *s*;
    int *old_setting* = *mp→selector*;

    mp→selector = *new_string*;
    mp_print_variable_name(*mp*, *mp→warning_info_node*);
    s = *mp_make_string*(*mp*);
    mp→selector = *old_setting*;
    mp_snprintf (*msg*, 256, "%s␣the␣definition␣of␣%s", *msg_start*, *s→str*);
    delete_str_ref(*s*);
  }
  set_cur_sym(*mp→frozen_end_def*);
  break;
case *op_defining*:
  {
    char *∗s* = *mp_str*(*mp*, *text*(*mp→warning_info*));

    mp_snprintf (*msg*, 256, "%s␣the␣definition␣of␣%s", *msg_start*, *s*);
  }
  set_cur_sym(*mp→frozen_end_def*);
  break;
case *loop_defining*:
  {
    char *∗s* = *mp_str*(*mp*, *text*(*mp→warning_info*));

    mp_snprintf (*msg*, 256, "%s␣the␣text␣of␣a␣%s␣loop", *msg_start*, *s*);
  }
  hlp[0] = "I␣suspect␣you␣have␣forgotten␣an␣'endfor',";
  set_cur_sym(*mp→frozen_end_for*);
  break;
This code is used in section 724.

726.   The *runaway* procedure displays the first part of the text that occurred when METAPOST began its special *scanner_status*, if that text has been saved.

⟨ Declarations 8 ⟩ +≡
  static void *mp_runaway*(**MP** *mp*);

727.   **void** *mp_runaway*(**MP** *mp*)
  {
    if (*mp*→*scanner_status* > *flushing*) {
      mp_print_nl(*mp*, "Runaway␣");
      switch (*mp*→*scanner_status*) {
      case *absorbing*: *mp_print*(*mp*, "text?");
        break;
      case *var_defining*: **case** *op_defining*: *mp_print*(*mp*, "definition?");
        break;
      case *loop_defining*: *mp_print*(*mp*, "loop?");
        break;
      }   /* there are no other cases */
      mp_print_ln(*mp*);
      mp_show_token_list(*mp*, *mp_link*(*mp*→*hold_head*), Λ, *mp*→*error_line* − 10, 0);
    }
  }

728.   We need to mention a procedure that may be called by *get_next*.

⟨ Declarations 8 ⟩ +≡
  static void *mp_firm_up_the_line*(**MP** *mp*);

729.   And now we're ready to take the plunge into *get_next* itself. Note that the behavior depends on the
scanner_status because percent signs and double quotes need to be passed over when skipping TeX material.

```
void mp_get_next(MP mp)
{    /* sets cur_cmd, cur_mod, cur_sym to next token */
  mp_sym cur_sym_;    /* speed up access */
RESTART: set_cur_sym(Λ);
  set_cur_sym_mod(0);
  if (file_state) {
    int k;    /* an index into buffer */
    ASCII_code c;    /* the current character in the buffer */
    int cclass;    /* its class number */    /* Input from external file; goto restart if no input found,
        or return if a non-symbolic token is found */    /* A percent sign appears in buffer[limit];
        this makes it unnecessary to have a special test for end-of-line. */
SWITCH: c = mp→buffer[loc];
    incr(loc);
    cclass = mp→char_class[c];
    switch (cclass) {
    case digit_class: scan_numeric_token((c − '0'));
      return;
      break;
    case period_class: cclass = mp→char_class[mp→buffer[loc]];
      if (cclass > period_class) {
        goto SWITCH;
      }
      else if (cclass < period_class) {    /* class = digit_class */
        scan_fractional_token(0);
        return;
      }
      break;
    case space_class: goto SWITCH;
      break;
    case percent_class:
      if (mp→scanner_status ≡ tex_flushing) {
        if (loc < limit) goto SWITCH;
      }    /* Move to next line of file, or goto restart if there is no next line */
      switch (move_to_next_line(mp)) {
      case 1: goto RESTART;
        break;
      case 2: goto COMMON_ENDING;
        break;
      default: break;
      }
      check_interrupt;
      goto SWITCH;
      break;
    case string_class:
      if (mp→scanner_status ≡ tex_flushing) {
        goto SWITCH;
      }
      else {
        if (mp→buffer[loc] ≡ '"') {
```

$set\_cur\_mod\_str(mp\_rts(mp, \texttt{""}));$
}
else {
$\quad k = loc;$
$\quad mp\text{-}buffer[limit + 1] = xord(\texttt{'"'});$
do {
$\quad incr(loc);$
} **while** $(mp\text{-}buffer[loc] \neq \texttt{'"'});$
if $(loc > limit)$ {   /∗ Decry the missing string delimiter and **goto** *restart* ∗/   /∗ We go
        to *restart* after this error message, not to SWITCH, because the *clear_for_error_prompt*
        routine might have reinstated *token_state* after **error** has finished. ∗/
$\quad$ **const char** $*hlp[\,] = \{\texttt{"Strings\_should\_finish\_on\_the\_same\_line\_as\_they\_began."},$
$\quad\quad \texttt{"I'␣ve␣deleted␣the␣partial␣string;␣you␣might␣want␣to"},$
$\quad\quad \texttt{"insert␣another␣by␣typing,␣e.g.,␣`I\"new␣string\"'."}, \Lambda\};$
$\quad loc = limit;$ /∗ the next character to be read on this line will be "%" ∗/
$\quad mp\_error(mp, \texttt{"Incomplete␣string␣token␣has␣been␣flushed"}, hlp, false);$
$\quad$ **goto** RESTART;
}
$str\_room((\textbf{size\_t})(loc - k));$
do {
$\quad append\_char(mp\text{-}buffer[k]);$
$\quad incr(k);$
} **while** $(k \neq loc);$
$set\_cur\_mod\_str(mp\_make\_string(mp));$
}
$incr(loc);$
$set\_cur\_cmd((mp\_variable\_type)mp\_string\_token);$
return;
}
break;
case *isolated_classes*: $k = loc - 1;$
$\quad$ **goto** FOUND;
$\quad$ **break**;
case *invalid_class*:
$\quad$ **if** $(mp\text{-}scanner\_status \equiv tex\_flushing)$ {
$\quad\quad$ **goto** SWITCH;
$\quad$ }
$\quad$ **else** {   /∗ Decry the invalid character and **goto** *restart* ∗/
$\quad\quad$ /∗ We go to *restart* instead of to SWITCH, because we might enter *token_state* after the error
$\quad\quad\quad$ has been dealt with (cf. *clear_for_error_prompt*). ∗/
$\quad\quad$ **const char** $*hlp[\,] = \{\texttt{"A␣funny␣symbol␣that␣I␣can\'␣t␣read␣has␣just␣been␣input."},$
$\quad\quad\quad \texttt{"Continue,␣and␣I'␣ll␣forget␣that␣it␣ever␣happened."}, \Lambda\};$
$\quad\quad mp\_error(mp, \texttt{"Text␣line␣contains␣an␣invalid␣character"}, hlp, false);$
$\quad\quad$ **goto** RESTART;
$\quad$ }
$\quad$ **break**;
default: **break**;   /∗ letters, etc. ∗/
}
$k = loc - 1;$
while $(mp\text{-}char\_class[mp\text{-}buffer[loc]] \equiv cclass)\ incr(loc);$
FOUND: $set\_cur\_sym(mp\_id\_lookup(mp, (\textbf{char} *)(mp\text{-}buffer + k), (\textbf{size\_t})(loc - k), true));$
}

else {   /∗ Input from token list; **goto** *restart* if end of list or if a parameter needs to be expanded, or **return** if a non-symbolic token is found ∗/

  if ($nloc \neq \Lambda \wedge mp\_type(nloc) \equiv mp\_symbol\_node$) {   /∗ symbolic token ∗/

    int $cur\_sym\_mod\_ = mp\_name\_type(nloc)$;

    halfword $cur\_info = mp\_sym\_info(nloc)$;

    $set\_cur\_sym(mp\_sym\_sym(nloc))$;

    $set\_cur\_sym\_mod(cur\_sym\_mod\_)$;

    $nloc = mp\_link(nloc)$;   /∗ move to next ∗/

    if ($cur\_sym\_mod\_ \equiv mp\_expr\_sym$) {

      $set\_cur\_cmd((mp\_variable\_type)mp\_capsule\_token)$;

      $set\_cur\_mod\_node(mp\text{-}param\_stack[param\_start + cur\_info])$;

      $set\_cur\_sym\_mod(0)$;

      $set\_cur\_sym(\Lambda)$;

      return;

    }

    else if ($cur\_sym\_mod\_ \equiv mp\_suffix\_sym \vee cur\_sym\_mod\_ \equiv mp\_text\_sym$) {

      $mp\_begin\_token\_list(mp, mp\text{-}param\_stack[param\_start + cur\_info], (\mathbf{quarterword})\ parameter)$;

      goto RESTART;

    }

  }

  else if ($nloc \neq \Lambda$) {   /∗ Get a stored numeric or string or capsule token and **return** ∗/

    if ($mp\_name\_type(nloc) \equiv mp\_token$) {

      if ($mp\_type(nloc) \equiv mp\_known$) {

        $set\_cur\_mod\_number(value\_number(nloc))$;

        $set\_cur\_cmd((mp\_variable\_type)mp\_numeric\_token)$;

      }

      else {

        $set\_cur\_mod\_str(value\_str(nloc))$;

        $set\_cur\_cmd((mp\_variable\_type)mp\_string\_token)$;

        $add\_str\_ref(cur\_mod\_str())$;

      }

    }

    else {

      $set\_cur\_mod\_node(nloc)$;

      $set\_cur\_cmd((mp\_variable\_type)mp\_capsule\_token)$;

    }

    $nloc = mp\_link(nloc)$;

    return;

  }

  else {   /∗ we are done with this token list ∗/

    $mp\_end\_token\_list(mp)$;

    goto RESTART;   /∗ resume previous level ∗/

  }

}

COMMON_ENDING: /∗ When a symbolic token is declared to be '**outer**', its command code is increased by *outer_tag*. ∗/

$cur\_sym\_ = cur\_sym()$;

$set\_cur\_cmd(eq\_type(cur\_sym\_))$;

$set\_cur\_mod(equiv(cur\_sym\_))$;

$set\_cur\_mod\_node(equiv\_node(cur\_sym\_))$;

if ($cur\_cmd() \geq mp\_outer\_tag$) {

  if ($mp\_check\_outer\_validity(mp)$) $set\_cur\_cmd(cur\_cmd() - mp\_outer\_tag)$;

  else goto RESTART;
  }
 }

730. The global variable *force_eof* is normally *false*; it is set *true* by an **endinput** command.

⟨ Global variables 14 ⟩ +≡
 boolean *force_eof*; /∗ should the next **input** be aborted early? ∗/

731. ⟨ Declarations 8 ⟩ +≡
 static int *move_to_next_line*(**MP** *mp*);

732.   **static int** *move_to_next_line*(**MP** *mp*){ **if** (*name* > *max_spec_src*) {
            /∗ Read next line of file into *buffer*, or return 1 (**goto** *restart*) if the file has ended ∗/
            /∗ We must decrement *loc* in order to leave the buffer in a valid state when an error condition
               causes us to **goto** *restart* without calling *end_file_reading*. ∗/
        { *incr* (**line**) ;
        mp→*first* = (**size_t**) *start*;
        if (¬*mp*→*force_eof*) {
          if (*mp_input_ln*(*mp*, *cur_file*))   /∗ not end of file ∗/
            mp_firm_up_the_line(*mp*);   /∗ this sets *limit* ∗/
          else *mp*→*force_eof* = *true*;
        }
        ;
        if (*mp*→*force_eof*) {
          mp→*force_eof* = *false*;
          decr(*loc*);
          if (*mpx_reading*) {   /∗ Complain that the MPX file ended unexpectly; then set *cur_sym*: =
                    mp→*frozen_mpx_break* and **goto** *comon_ending* ∗/
               /∗ We should never actually come to the end of an MPX file because such files should have an
                  mpxbreak after the translation of the last **btex** . . . **etex** block. ∗/
            const char ∗*hlp*[ ] = {"The␣file␣had␣too␣few␣picture␣expressions␣for␣btex...etex",
                "blocks.␣␣Such␣files␣are␣normally␣generated␣automatically",
                "but␣this␣one␣got␣messed␣up.␣␣You␣might␣want␣to␣insert␣a",
                "picture␣expression␣now.", Λ};
            mp→*mpx_name*[*iindex*] = *mpx_finished*;
            mp_error(*mp*, "mpx␣file␣ended␣unexpectedly", *hlp*, *false*);
            set_cur_sym(*mp*→*frozen_mpx_break*);
            return 2;
          }
          else {
            mp_print_char(*mp*, *xord*(')'));
            decr(*mp*→*open_parens*);
            update_terminal( );   /∗ show user that file has been read ∗/
            mp_end_file_reading(*mp*);   /∗ resume previous level ∗/
            if (*mp_check_outer_validity*(*mp*)) **return** 1;
            else **return** 1;
          }
        }
        mp→*buffer*[*limit*] = *xord*('%');
        mp→*first* = (**size_t**)(*limit* + 1);
        loc = *start*;   /∗ ready to read ∗/
        } }
        else {
          if (*mp*→*input_ptr* > 0) {   /∗ text was inserted during error recovery or by **scantokens** ∗/
            mp_end_file_reading(*mp*);   /∗ goto RESTART ∗/
            return 1;   /∗ resume previous level ∗/
          }
          if (*mp*→*job_name* ≡ Λ ∧ (*mp*→*selector* < *log_only* ∨ *mp*→*selector* ≥ *write_file*)) *mp_open_log_file*(*mp*);
          if (*mp*→*interaction* > *mp_nonstop_mode*) {
            if (*limit* ≡ *start*)   /∗ previous line was empty ∗/
              mp_print_nl(*mp*, "(Please␣type␣a␣command␣or␣say␣'end')");
            mp_print_ln(*mp*);
            mp→*first* = (**size_t**) *start*;

$prompt\_input("*");$ /∗ input on-line into $buffer$ ∗/
$limit = (\textbf{halfword})\ mp\text{-}last;$
$mp\text{-}buffer[limit] = xord('%');$
$mp\text{-}first = (\textbf{size\_t})(limit + 1);$
$loc = start;$
}
else {
$mp\_fatal\_error(mp, "***\textsubscript{⊔}(job\textsubscript{⊔}aborted,\textsubscript{⊔}no\textsubscript{⊔}legal\textsubscript{⊔}end\textsubscript{⊔}found)");$ /∗ nonstop mode,
which is intended for overnight batch processing, never waits for on-line input ∗/
}
}
return 0; }

733.   If the user has set the $mp\_pausing$ parameter to some positive value, and if nonstop mode has not
been selected, each line of input is displayed on the terminal and the transcript file, followed by '=>'. META-
POST waits for a response. If the response is NULL (i.e., if nothing is typed except perhaps a few blank
spaces), the original line is accepted as it stands; otherwise the line typed is used instead of the line in the
file.

void $mp\_firm\_up\_the\_line(\textbf{MP}\ mp)$
{
size_t $k;$ /∗ an index into $buffer$ ∗/
$limit = (\textbf{halfword})\ mp\text{-}last;$
if $((\neg mp\text{-}noninteractive) \wedge (number\_positive(internal\_value(mp\_pausing))) \wedge (mp\text{-}interaction >$
$mp\_nonstop\_mode))$ {
$wake\_up\_terminal();$
$mp\_print\_ln(mp);$
if $(start < limit)$ {
for $(k = (\textbf{size\_t})\ start;\ k < (\textbf{size\_t})\ limit;\ k{+}{+})$ {
$mp\_print\_char(mp, mp\text{-}buffer[k]);$
}
}
$mp\text{-}first = (\textbf{size\_t})\ limit;$
$prompt\_input("=>");$ /∗ wait for user response ∗/
;
if $(mp\text{-}last > mp\text{-}first)$ {
for $(k = mp\text{-}first;\ k < mp\text{-}last;\ k{+}{+})$ {   /∗ move line down in buffer ∗/
$mp\text{-}buffer[k + (\textbf{size\_t})\ start - mp\text{-}first] = mp\text{-}buffer[k];$
}
$limit = (\textbf{halfword})((\textbf{size\_t})\ start + mp\text{-}last - mp\text{-}first);$
}
}
}

734.   **Dealing with TEX material.**   The **btex** . . . **etex** and **verbatimtex** . . . **etex** features need to
be implemented at a low level in the scanning process so that METAPOST can stay in synch with the a
preprocessor that treats blocks of TEX material as they occur in the input file without trying to expand
METAPOST macros. Thus we need a special version of *get_next* that does not expand macros and such but
does handle **btex**, **verbatimtex**, etc.

The special version of *get_next* is called *get_t_next*. It works by flushing **btex** . . . **etex** and **verbatimtex**
. . . **etex** blocks, switching to the MPX file when it sees **btex**, and switching back when it sees **mpxbreak**.

#**define**   *btex_code*   0
#**define**   *verbatim_code*   1

735.   ⟨ Put each of METAPOST's primitives into the hash table 200 ⟩ +≡
  mp_primitive(*mp*, "btex", *mp_start_tex*, *btex_code*);
  ;
  mp_primitive(*mp*, "verbatimtex", *mp_start_tex*, *verbatim_code*);
  ;
  mp_primitive(*mp*, "etex", *mp_etex_marker*, 0);
  mp→*frozen_etex* = *mp_frozen_primitive*(*mp*, "etex", *mp_etex_marker*, 0);
  ;
  mp_primitive(*mp*, "mpxbreak", *mp_mpx_break*, 0);
  mp→*frozen_mpx_break* = *mp_frozen_primitive*(*mp*, "mpxbreak", *mp_mpx_break*, 0);

736.   ⟨ Cases of *print_cmd_mod* for symbolic printing of primitives 233 ⟩ +≡
case *mp_start_tex*:
  if (*m* ≡ *btex_code*) *mp_print*(*mp*, "btex");
  else *mp_print*(*mp*, "verbatimtex");
  break;
case *mp_etex_marker*: *mp_print*(*mp*, "etex");
  break;
case *mp_mpx_break*: *mp_print*(*mp*, "mpxbreak");
  break;

737.   Actually, *get_t_next* is a macro that avoids procedure overhead except in the unusual case where
btex, **verbatimtex**, **etex**, or **mpxbreak** is encountered.

#**define**   *get_t_next*(*a*)   **do**
        {
          mp_get_next(*mp*);
          if (*cur_cmd*( ) ≤ *mp_max_pre_command*) *mp_t_next*(*mp*);
        }
        while (0)

738.   ⟨ Declarations 8 ⟩ +≡
  static void *mp_t_next*(**MP** *mp*);
  static void *mp_start_mpx_input*(**MP** *mp*);

739.   **static void** $mp\_t\_next(\mathbf{MP}\ mp)$
{
  int old_status;   /∗ saves the $scanner\_status$ ∗/
  integer old_info;   /∗ saves the $warning\_info$ ∗/
  while $(cur\_cmd(\,) \leq mp\_max\_pre\_command)$ {
    if $(cur\_cmd(\,) \equiv mp\_mpx\_break)$ {
      if $(\neg file\_state \lor (mp\text{→}mpx\_name[iindex] \equiv absent))$ {
        ⟨ Complain about a misplaced **mpxbreak** 743 ⟩;
      }
      else {
        $mp\_end\_mpx\_reading(mp)$;
        goto TEX_FLUSH;
      }
    }
    else if $(cur\_cmd(\,) \equiv mp\_start\_tex)$ {
      if $(token\_state \lor (name \leq max\_spec\_src))$ {
        ⟨ Complain that we are not reading a file 742 ⟩;
      }
      else if $(mpx\_reading)$ {
        ⟨ Complain that MPX files cannot contain TEX material 741 ⟩;
      }
      else if $((cur\_mod(\,) \neq verbatim\_code) \land (mp\text{→}mpx\_name[iindex] \neq mpx\_finished))$ {
        if $(\neg mp\_begin\_mpx\_reading(mp))$ $mp\_start\_mpx\_input(mp)$;
      }
      else {
        goto TEX_FLUSH;
      }
    }
    else {
      ⟨ Complain about a misplaced **etex** 744 ⟩;
    }
    goto COMMON_ENDING;
  TEX_FLUSH: ⟨ Flush the TEX material 740 ⟩;
  COMMON_ENDING: $mp\_get\_next(mp)$;
  }
}

740.   We could be in the middle of an operation such as skipping false conditional text when TEX material
is encountered, so we must be careful to save the $scanner\_status$.

⟨ Flush the TEX material 740 ⟩ ≡
  $old\_status = mp\text{→}scanner\_status$;
  $old\_info = mp\text{→}warning\_line$;
  $mp\text{→}scanner\_status = tex\_flushing$; $mp\text{→}warning\_line = \mathbf{line}$;
  do {
    $mp\_get\_next(mp)$;
  } **while** $(cur\_cmd(\,) \neq mp\_etex\_marker)$;
  $mp\text{→}scanner\_status = old\_status$; $mp\text{→}warning\_line = old\_info$

This code is used in section 739.

741.   ⟨Complain that `MPX` files cannot contain TEX material 741⟩ ≡
```
  {
    const char *hlp[] = {"This␣file␣contains␣picture␣expressions␣for␣btex...etex",
        "blocks.␣␣Such␣files␣are␣normally␣generated␣automatically",
        "but␣this␣one␣seems␣to␣be␣messed␣up.␣␣I'll␣just␣keep␣going",
        "and␣hope␣for␣the␣best.",Λ};
    mp_error(mp, "An␣mpx␣file␣cannot␣contain␣btex␣or␣verbatimtex␣blocks", hlp, true);
  }
```
This code is used in section 739.

742.   ⟨Complain that we are not reading a file 742⟩ ≡
```
  {
    const char *hlp[] = {"I'll␣have␣to␣ignore␣this␣preprocessor␣command␣because␣it",
        "only␣works␣when␣there␣is␣a␣file␣to␣preprocess.␣␣You␣might",
        "want␣to␣delete␣everything␣up␣to␣the␣next␣'etex'.",Λ};
    mp_error(mp, "You␣can␣only␣use␣'btex'␣or␣'verbatimtex'␣in␣a␣file", hlp, true);
  }
```
This code is used in section 739.

743.   ⟨Complain about a misplaced **mpxbreak** 743⟩ ≡
```
  {
    const char *hlp[] = {"I'll␣ignore␣this␣preprocessor␣command␣because␣it",
        "doesn't␣belong␣here",Λ};
    mp_error(mp, "Misplaced␣mpxbreak", hlp, true);
  }
```
This code is used in section 739.

744.   ⟨Complain about a misplaced **etex** 744⟩ ≡
```
  {
    const char *hlp[] = {"There␣is␣no␣btex␣or␣verbatimtex␣for␣this␣to␣match",Λ};
    mp_error(mp, "Extra␣etex␣will␣be␣ignored", hlp, true);
  }
```
This code is used in section 739.

745.   Scanning macro definitions.   METAPOST has a variety of ways to tuck tokens away into token lists for later use: Macros can be defined with **def**, **vardef**, **primarydef**, etc.; repeatable code can be defined with **for**, **forever**, **forsuffixes**. All such operations are handled by the routines in this part of the program.

The modifier part of each command code is zero for the "ending delimiters" like **enddef** and **endfor**.

#**define** $start\_def$ 1   /∗ command modifier for **def** ∗/
#**define** var_def 2   /∗ command modifier for **vardef** ∗/
#**define** end_def 0   /∗ command modifier for **enddef** ∗/
#**define** $start\_forever$ 1   /∗ command modifier for **forever** ∗/
#**define** $start\_for$ 2   /∗ command modifier for **forever** ∗/
#**define** $start\_forsuffixes$ 3   /∗ command modifier for **forever** ∗/
#**define** end_for 0   /∗ command modifier for **endfor** ∗/

⟨ Put each of METAPOST's primitives into the hash table 200 ⟩ +≡
  $mp\_primitive(mp, \texttt{"def"}, mp\_macro\_def, start\_def);$
  ;
  $mp\_primitive(mp, \texttt{"vardef"}, mp\_macro\_def, var\_def);$
  ;
  $mp\_primitive(mp, \texttt{"primarydef"}, mp\_macro\_def, mp\_secondary\_primary\_macro);$
  ;
  $mp\_primitive(mp, \texttt{"secondarydef"}, mp\_macro\_def, mp\_tertiary\_secondary\_macro);$
  ;
  $mp\_primitive(mp, \texttt{"tertiarydef"}, mp\_macro\_def, mp\_expression\_tertiary\_macro);$
  ;
  $mp\_primitive(mp, \texttt{"enddef"}, mp\_macro\_def, end\_def);$
  $mp \neg frozen\_end\_def = mp\_frozen\_primitive(mp, \texttt{"enddef"}, mp\_macro\_def, end\_def);$
  ;
  $mp\_primitive(mp, \texttt{"for"}, mp\_iteration, start\_for);$
  ;
  $mp\_primitive(mp, \texttt{"forsuffixes"}, mp\_iteration, start\_forsuffixes);$
  ;
  $mp\_primitive(mp, \texttt{"forever"}, mp\_iteration, start\_forever);$
  ;
  $mp\_primitive(mp, \texttt{"endfor"}, mp\_iteration, end\_for);$
  $mp \neg frozen\_end\_for = mp\_frozen\_primitive(mp, \texttt{"endfor"}, mp\_iteration, end\_for);$

746.   ⟨Cases of *print_cmd_mod* for symbolic printing of primitives 233⟩ +≡
case *mp_macro_def*:
 if $(m \leq var\_def)$ {
  if $(m \equiv start\_def)$ *mp_print*(*mp*, "def");
  else if $(m < start\_def)$ *mp_print*(*mp*, "enddef");
  else *mp_print*(*mp*, "vardef");
 }
 else if $(m \equiv mp\_secondary\_primary\_macro)$ {
  mp_print(*mp*, "primarydef");
 }
 else if $(m \equiv mp\_tertiary\_secondary\_macro)$ {
  mp_print(*mp*, "secondarydef");
 }
 else {
  mp_print(*mp*, "tertiarydef");
 }
 break;
case *mp_iteration*:
 if $(m \equiv start\_forever)$ *mp_print*(*mp*, "forever");
 else if $(m \equiv end\_for)$ *mp_print*(*mp*, "endfor");
 else if $(m \equiv start\_for)$ *mp_print*(*mp*, "for");
 else *mp_print*(*mp*, "forsuffixes");
 break;

747.   Different macro-absorbing operations have different syntaxes, but they also have a lot in common. There is a list of special symbols that are to be replaced by parameter tokens; there is a special command code that ends the definition; the quotation conventions are identical. Therefore it makes sense to have most of the work done by a single subroutine. That subroutine is called *scan_toks*.

The first parameter to *scan_toks* is the command code that will terminate scanning (either *macro_def* or *iteration*).

The second parameter, *subst_list*, points to a (possibly empty) list of non-symbolic nodes whose *info* and *value* fields specify symbol tokens before and after replacement. The list will be returned to free storage by *scan_toks*.

The third parameter is simply appended to the token list that is built. And the final parameter tells how many of the special operations #@!, @!, and @!# are to be replaced by suffix parameters. When such parameters are present, they are called (SUFFIX0), (SUFFIX1), and (SUFFIX2).

⟨Types in the outer block 33⟩ +≡
 typedef struct mp_subst_list_item {
  mp_name_type_type *info_mod*;
  quarterword *value_mod*;
  mp_sym *info*;
  halfword *value_data*;
  struct mp_subst_list_item *∗link*;
 } **mp_subst_list_item**;

748.

```
static mp_node mp_scan_toks(MP mp, mp_command_code terminator, mp_subst_list_item
        *subst_list, mp_node tail_end, quarterword suffix_count)
{
  mp_node p;        /* tail of the token list being built */
  mp_subst_list_item *q = Λ;      /* temporary for link management */
  integer balance;      /* left delimiters minus right delimiters */
  halfword cur_data;
  quarterword cur_data_mod = 0;

  p = mp→hold_head;
  balance = 1;
  mp_link(mp→hold_head) = Λ;
  while (1) {
    get_t_next(mp);
    cur_data = −1;
    if (cur_sym() ≠ Λ) {
      ⟨Substitute for cur_sym, if it's on the subst_list 751⟩;
      if (cur_cmd() ≡ terminator) {
        ⟨Adjust the balance; break if it's zero 752⟩;
      }
      else if (cur_cmd() ≡ mp_macro_special) {      /* Handle quoted symbols, #@!, @!, or @!# */
        if (cur_mod() ≡ quote) {
          get_t_next(mp);
        }
        else if (cur_mod() ≤ suffix_count) {
          cur_data = cur_mod() − 1;
          cur_data_mod = mp_suffix_sym;
        }
      }
    }
    if (cur_data ≠ −1) {
      mp_node pp = mp_get_symbolic_node(mp);

      set_mp_sym_info(pp, cur_data);
      mp_name_type(pp) = cur_data_mod;
      mp_link(p) = pp;
    }
    else {
      mp_link(p) = mp_cur_tok(mp);
    }
    p = mp_link(p);
  }
  mp_link(p) = tail_end;
  while (subst_list) {
    q = subst_list→link;
    xfree(subst_list);
    subst_list = q;
  }
  return mp_link(mp→hold_head);
}
```

749.

  void *mp_print_sym*(**mp_sym** *sym*)
  {
    printf("{type␣=␣%d,␣v␣=␣{type␣=␣%d,␣data␣=␣{indep␣=␣{scale␣=␣%d,␣serial␣=␣%d},␣n␣=␣%d,␣\
      str␣=␣%p,␣sym␣=␣%p,␣node␣=␣%p,␣p␣=␣%p}},␣text␣=␣%p}\n", *sym→type*, *sym→v.type*, (**int**)
      sym→v.data.indep.scale, (**int**) *sym→v.data.indep.serial*, *sym→v.data.n.type*, *sym→v.data.str*,
      sym→v.data.sym, *sym→v.data.node*, *sym→v.data.p*, *sym→text*);
    if (*is_number*(*sym→v.data.n*)) {
      mp_number *n* = *sym→v.data.n*;
      printf("{data␣=␣{dval␣=␣%f,␣val␣=␣%d},␣type␣=␣%d}\n", *n.data.dval*, *n.data.val*, *n.type*);
    }
    if (*sym→text* ≠ Λ) {
      mp_string *t* = *sym→text*;
      printf("{str␣=␣%p␣\"%s\",␣len␣=␣%d,␣refs␣=␣%d}\n", *t→str*, *t→str*, (**int**) *t→len*, *t→refs*);
    }
  }

750.

⟨ Declarations 8 ⟩ +≡
  void *mp_print_sym*(**mp_sym** *sym*);

751. ⟨ Substitute for *cur_sym*, if it's on the *subst_list* 751 ⟩ ≡
  {
    q = *subst_list*;
    while (*q* ≠ Λ) {
      if (*q→info* ≡ *cur_sym*() ∧ *q→info_mod* ≡ *cur_sym_mod*()) {
        cur_data = *q→value_data*;
        cur_data_mod = *q→value_mod*;
        set_cur_cmd((*mp_variable_type*)*mp_relax*);
        break;
      }
      q = *q→link*;
    }
  }

This code is used in section 748.

752. ⟨ Adjust the balance; **break** if it's zero 752 ⟩ ≡
  if (*cur_mod*() > 0) {
    incr(*balance*);
  }
  else {
    decr(*balance*);
    if (*balance* ≡ 0) **break**;
  }

This code is used in section 748.

753.   Four commands are intended to be used only within macro texts: **quote**, `#@!`, `@!`, and `@!#`. They are variants of a single command code called *macro_special*.

#define   *quote*   0   /∗ *macro_special* modifier for **quote** ∗/
#define   *macro_prefix*   1   /∗ *macro_special* modifier for `#@!` ∗/
#define   *macro_at*   2   /∗ *macro_special* modifier for `@!` ∗/
#define   *macro_suffix*   3   /∗ *macro_special* modifier for `@!#` ∗/

⟨ Put each of METAPOST's primitives into the hash table 200 ⟩ +≡
  mp_primitive (*mp*, "quote", *mp_macro_special*, *quote*);
  ;
  mp_primitive (*mp*, "#@", *mp_macro_special*, *macro_prefix*);
  ;
  mp_primitive (*mp*, "@", *mp_macro_special*, *macro_at*);
  ;
  mp_primitive (*mp*, "@#", *mp_macro_special*, *macro_suffix*);

754.   ⟨ Cases of *print_cmd_mod* for symbolic printing of primitives 233 ⟩ +≡
case *mp_macro_special* :
  switch (*m*) {
  case *macro_prefix* : *mp_print* (*mp*, "#@");
    break;
  case *macro_at* : *mp_print_char* (*mp*, *xord* ('@'));
    break;
  case *macro_suffix* : *mp_print* (*mp*, "@#");
    break;
  default: *mp_print* (*mp*, "quote");
    break;
  }
  break;

755.   Here is a routine that's used whenever a token will be redefined. If the user's token is unredefinable, the '*mp→frozen_inaccessible*' token is substituted; the latter is redefinable but essentially impossible to use, hence METAPOST's tables won't get fouled up.

  static void *mp_get_symbol* (**MP** *mp*)
  {   /∗ sets *cur_sym* to a safe symbol ∗/
  RESTART: *get_t_next* (*mp*);
    if ((*cur_sym* ( ) ≡ Λ) ∨ *mp_is_frozen* (*mp*, *cur_sym* ( ))) {
      const char ∗*hlp* [ ] = {"Sorry:␣You␣can\'t␣redefine␣a␣number,␣string,␣or␣expr.",
          "I'␣ve␣inserted␣an␣inaccessible␣symbol␣so␣that␣your",
          "definition␣will␣be␣completed␣without␣mixing␣me␣up␣too␣badly.", Λ};
      if (*cur_sym* ( ) ≠ Λ) *hlp* [0] = "Sorry:␣You␣can\'t␣redefine␣my␣error−recovery␣tokens.";
      else if (*cur_cmd* ( ) ≡ *mp_string_token*) *delete_str_ref* (*cur_mod_str* ( ));
      set_cur_sym (*mp→frozen_inaccessible*);
      mp_ins_error (*mp*, "Missing␣symbolic␣token␣inserted", *hlp*, *true*);
      ;
      goto RESTART;
    }
  }

756.   Before we actually redefine a symbolic token, we need to clear away its former value, if it was a variable. The following stronger version of *get_symbol* does that.

static void *mp_get_clear_symbol*(**MP** *mp*)
{
  mp_get_symbol(*mp*);
  mp_clear_symbol(*mp*, *cur_sym*( ), *false*);
}

757.   Here's another little subroutine; it checks that an equals sign or assignment sign comes along at the proper place in a macro definition.

static void *mp_check_equals*(**MP** *mp*)
{
  if (*cur_cmd*( ) ≠ *mp_equals*)
    if (*cur_cmd*( ) ≠ *mp_assignment*) {
      const char *∗hlp*[ ] = {"The␣next␣thing␣in␣this␣`def'␣should␣have␣been␣`='",
        "because␣I've␣already␣looked␣at␣the␣definition␣heading.",
        "But␣don't␣worry;␣I'll␣pretend␣that␣an␣equals␣sign",
        "was␣present.␣Everything␣from␣here␣to␣`enddef'",
        "will␣be␣the␣replacement␣text␣of␣this␣macro.", Λ};
      mp_back_error(*mp*, "Missing␣`='␣has␣been␣inserted", *hlp*, *true*);
      ;
    }
}

758.   A **primarydef**, **secondarydef**, or **tertiarydef** is rather easily handled now that we have *scan_toks*.
In this case there are two parameters, which will be `EXPR0` and `EXPR1`.

```
static void mp_make_op_def(MP mp)
{
  mp_command_code m;      /* the type of definition */

  mp_node q, r;      /* for list manipulation */
  mp_subst_list_item *qm = Λ, *qn = Λ;

  m = cur_mod( );
  mp_get_symbol(mp);
  qm = xmalloc(1, sizeof(mp_subst_list_item));
  qm→link = Λ;
  qm→info = cur_sym( );
  qm→info_mod = cur_sym_mod( );
  qm→value_data = 0;
  qm→value_mod = mp_expr_sym;
  mp_get_clear_symbol(mp);
  mp→warning_info = cur_sym( );
  mp_get_symbol(mp);
  qn = xmalloc(1, sizeof(mp_subst_list_item));
  qn→link = qm;
  qn→info = cur_sym( );
  qn→info_mod = cur_sym_mod( );
  qn→value_data = 1;
  qn→value_mod = mp_expr_sym;
  get_t_next(mp);
  mp_check_equals(mp);
  mp→scanner_status = op_defining;
  q = mp_get_symbolic_node(mp);
  set_ref_count(q, 0);
  r = mp_get_symbolic_node(mp);
  mp_link(q) = r;
  set_mp_sym_info(r, mp_general_macro);
  mp_name_type(r) = mp_macro_sym;
  mp_link(r) = mp_scan_toks(mp, mp_macro_def, qn, Λ, 0);
  mp→scanner_status = normal;
  set_eq_type(mp→warning_info, m);
  set_equiv_node(mp→warning_info, q);
  mp_get_x_next(mp);
}
```

759.   Parameters to macros are introduced by the keywords **expr**, **suffix**, **text**, **primary**, **secondary**, and **tertiary**.

⟨Put each of METAPOST's primitives into the hash table 200⟩ +≡

 $mp\_primitive(mp, \texttt{"expr"}, mp\_param\_type, mp\_expr\_param)$;

 ;

 $mp\_primitive(mp, \texttt{"suffix"}, mp\_param\_type, mp\_suffix\_param)$;

 ;

 $mp\_primitive(mp, \texttt{"text"}, mp\_param\_type, mp\_text\_param)$;

 ;

 $mp\_primitive(mp, \texttt{"primary"}, mp\_param\_type, mp\_primary\_macro)$;

 ;

 $mp\_primitive(mp, \texttt{"secondary"}, mp\_param\_type, mp\_secondary\_macro)$;

 ;

 $mp\_primitive(mp, \texttt{"tertiary"}, mp\_param\_type, mp\_tertiary\_macro)$;

760.   ⟨Cases of *print_cmd_mod* for symbolic printing of primitives 233⟩ +≡

case mp_param_type:

 if $(m \equiv mp\_expr\_param)$ $mp\_print(mp, \texttt{"expr"})$;

 else if $(m \equiv mp\_suffix\_param)$ $mp\_print(mp, \texttt{"suffix"})$;

 else if $(m \equiv mp\_text\_param)$ $mp\_print(mp, \texttt{"text"})$;

 else if $(m \equiv mp\_primary\_macro)$ $mp\_print(mp, \texttt{"primary"})$;

 else if $(m \equiv mp\_secondary\_macro)$ $mp\_print(mp, \texttt{"secondary"})$;

 else $mp\_print(mp, \texttt{"tertiary"})$;

 break;

761.   Let's turn next to the more complex processing associated with **def** and **vardef**. When the following procedure is called, *cur_mod* should be either *start_def* or *var_def*.

Note that although the macro scanner allows $def = : = enddef$ and $def : == enddef$; $def == enddef$ and $def : = : = enddef$ will generate an error because by the time the second of the two identical tokens is seen, its meaning has already become undefined.

static void *mp_scan_def* (**MP** *mp*)
{
 int *m*; /∗ the type of definition ∗/
 int *n*; /∗ the number of special suffix parameters ∗/
 int *k*; /∗ the total number of parameters ∗/
 int *c*; /∗ the kind of macro we're defining ∗/
 mp_subst_list_item *∗r* = Λ, *∗rp* = Λ; /∗ parameter-substitution list ∗/
 mp_node *q*; /∗ tail of the macro token list ∗/
 mp_node *p*; /∗ temporary storage ∗/
 quarterword *sym_type*; /∗ *expr_sym*, *suffix_sym*, or *text_sym* ∗/
 mp_sym *l_delim*, *r_delim*; /∗ matching delimiters ∗/
 $m = cur\_mod(\,)$;
 $c = mp\_general\_macro$;
 $mp\_link(mp\text{→}hold\_head) = \Lambda$;
 $q = mp\_get\_symbolic\_node(mp)$;
 $set\_ref\_count(q, 0)$;
 $r = \Lambda$; /∗ Scan the token or variable to be defined; set *n*, *scanner_status*, and *warning_info* ∗/
 if $(m \equiv start\_def)$ {
  $mp\_get\_clear\_symbol(mp)$;
  $mp\text{→}warning\_info = cur\_sym(\,)$;
  $get\_t\_next(mp)$;
  $mp\text{→}scanner\_status = op\_defining$;
  $n = 0$;
  $set\_eq\_type(mp\text{→}warning\_info, mp\_defined\_macro)$;
  $set\_equiv\_node(mp\text{→}warning\_info, q)$;
 }
 else { /∗ *var_def* ∗/
  $p = mp\_scan\_declared\_variable(mp)$;
  $mp\_flush\_variable(mp, equiv\_node(mp\_sym\_sym(p)), mp\_link(p), true)$;
  $mp\text{→}warning\_info\_node = mp\_find\_variable(mp, p)$;
  $mp\_flush\_node\_list(mp, p)$;
  if $(mp\text{→}warning\_info\_node \equiv \Lambda)$ { /∗ Change to 'a bad variable' ∗/
   const char *∗hlp*[ ] = {"After␣'vardef␣a'␣you␣can\'t␣say␣'vardef␣a.b'.",
    "So␣I'll␣have␣to␣discard␣this␣definition.", Λ};

   $mp\_error(mp, \texttt{"This␣variable␣already␣starts␣with␣a␣macro"}, hlp, true)$;
   $mp\text{→}warning\_info\_node = mp\text{→}bad\_vardef$;
  }
  $mp\text{→}scanner\_status = var\_defining$;
  $n = 2$;
  if $(cur\_cmd(\,) \equiv mp\_macro\_special \wedge cur\_mod(\,) \equiv macro\_suffix)$ { /∗ @!# ∗/
   $n = 3$;
   $get\_t\_next(mp)$;
  }
  $mp\_type(mp\text{→}warning\_info\_node) = (\textbf{quarterword})(mp\_unsuffixed\_macro - 2 + n)$;
   /∗ *mp_suffixed_macro* = *mp_unsuffixed_macro* + 1 ∗/
  $set\_value\_node(mp\text{→}warning\_info\_node, q)$;

```
  }
  k = n;
  if (cur_cmd( ) ≡ mp_left_delimiter) {
    /∗ Absorb delimited parameters, putting them into lists q and r ∗/
    do {
      l_delim = cur_sym( );
      r_delim = equiv_sym(cur_sym( ));
      get_t_next(mp);
      if ((cur_cmd( ) ≡ mp_param_type) ∧ (cur_mod( ) ≡ mp_expr_param)) {
        sym_type = mp_expr_sym;
      }
      else if ((cur_cmd( ) ≡ mp_param_type) ∧ (cur_mod( ) ≡ mp_suffix_param)) {
        sym_type = mp_suffix_sym;
      }
      else if ((cur_cmd( ) ≡ mp_param_type) ∧ (cur_mod( ) ≡ mp_text_param)) {
        sym_type = mp_text_sym;
      }
      else {
        const char ∗hlp[ ] = {"You␣should've␣had␣`expr'␣or␣`suffix'␣or␣`text'␣here.", Λ};
        mp_back_error(mp, "Missing␣parameter␣type;␣`expr'␣will␣be␣assumed", hlp, true);
        sym_type = mp_expr_sym;
      }   /∗ Absorb parameter tokens for type sym_type ∗/
      do {
        mp_link(q) = mp_get_symbolic_node(mp);
        q = mp_link(q);
        mp_name_type(q) = sym_type;
        set_mp_sym_info(q, k);
        mp_get_symbol(mp);
        rp = xmalloc(1, sizeof(mp_subst_list_item));
        rp→link = Λ;
        rp→value_data = k;
        rp→value_mod = sym_type;
        rp→info = cur_sym( );
        rp→info_mod = cur_sym_mod( );
        mp_check_param_size(mp, k);
        incr(k);
        rp→link = r;
        r = rp;
        get_t_next(mp);
      } while (cur_cmd( ) ≡ mp_comma);
      mp_check_delimiter(mp, l_delim, r_delim);
      get_t_next(mp);
    } while (cur_cmd( ) ≡ mp_left_delimiter);
  }
  if (cur_cmd( ) ≡ mp_param_type) {      /∗ Absorb undelimited parameters, putting them into list r ∗/
    rp = xmalloc(1, sizeof(mp_subst_list_item));
    rp→link = Λ;
    rp→value_data = k;
    if (cur_mod( ) ≡ mp_expr_param) {
      rp→value_mod = mp_expr_sym;
      c = mp_expr_macro;
    }
```

```
        else if (cur_mod( ) ≡ mp_suffix_param) {
          rp→value_mod = mp_suffix_sym;
          c = mp_suffix_macro;
        }
        else if (cur_mod( ) ≡ mp_text_param) {
          rp→value_mod = mp_text_sym;
          c = mp_text_macro;
        }
        else {
          c = cur_mod( );
          rp→value_mod = mp_expr_sym;
        }
        mp_check_param_size(mp, k);
        incr(k);
        mp_get_symbol(mp);
        rp→info = cur_sym( );
        rp→info_mod = cur_sym_mod( );
        rp→link = r;
        r = rp;
        get_t_next(mp);
        if (c ≡ mp_expr_macro) {
          if (cur_cmd( ) ≡ mp_of_token) {
            c = mp_of_macro;
            rp = xmalloc(1, sizeof(mp_subst_list_item));
            rp→link = Λ;
            mp_check_param_size(mp, k);
            rp→value_data = k;
            rp→value_mod = mp_expr_sym;
            mp_get_symbol(mp);
            rp→info = cur_sym( );
            rp→info_mod = cur_sym_mod( );
            rp→link = r;
            r = rp;
            get_t_next(mp);
          }
        }
      }
      mp_check_equals(mp);
      p = mp_get_symbolic_node(mp);
      set_mp_sym_info(p, c);
      mp_name_type(p) = mp_macro_sym;
      mp_link(q) = p;       /* Attach the replacement text to the tail of node p */
        /* We don't put 'mp→frozen_end_group' into the replacement text of a vardef, because the user may
           want to redefine 'endgroup'. */
      if (m ≡ start_def) {
        mp_link(p) = mp_scan_toks(mp, mp_macro_def, r, Λ, (quarterword) n);
      }
      else {
        mp_node qq = mp_get_symbolic_node(mp);

        set_mp_sym_sym(qq, mp→bg_loc);
        mp_link(p) = qq;
        p = mp_get_symbolic_node(mp);
```

$set\_mp\_sym\_sym(p, mp\text{-}eg\_loc);$
$mp\_link(qq) = mp\_scan\_toks(mp, mp\_macro\_def, r, p, (\textbf{quarterword}) \, n);$
    }
    if $(mp\text{-}warning\_info\_node \equiv mp\text{-}bad\_vardef) \;\; mp\_flush\_token\_list(mp, value\_node(mp\text{-}bad\_vardef));$
    $mp\text{-}scanner\_status = normal;$
    $mp\_get\_x\_next(mp);$
}

762.   ⟨Global variables 14⟩ +≡
  mp_sym $bg\_loc;$
  mp_sym $eg\_loc;$ /∗ hash addresses of 'begingroup' and 'endgroup' ∗/

763.   ⟨Initialize table entries 182⟩ +≡
  $mp\text{-}bad\_vardef = mp\_get\_value\_node(mp);$
  $mp\_name\_type(mp\text{-}bad\_vardef) = mp\_root;$
  $set\_value\_sym(mp\text{-}bad\_vardef, mp\text{-}frozen\_bad\_vardef);$

764.   ⟨Free table entries 183⟩ +≡
  $mp\_free\_value\_node(mp, mp\text{-}bad\_vardef);$

765.   Expanding the next token.   Only a few command codes $<$ *min_command* can possibly be returned by *get_t_next*; in increasing order, they are *if_test*, *fi_or_else*, *input*, *iteration*, *repeat_loop*, *exit_test*, *relax*, *scan_tokens*, *expand_after*, and *defined_macro*.

METAPOST usually gets the next token of input by saying *get_x_next*. This is like *get_t_next* except that it keeps getting more tokens until finding *cur_cmd* $\geq$ *min_command*. In other words, *get_x_next* expands macros and removes conditionals or iterations or input instructions that might be present.

It follows that *get_x_next* might invoke itself recursively. In fact, there is massive recursion, since macro expansion can involve the scanning of arbitrarily complex expressions, which in turn involve macro expansion and conditionals, etc.

Therefore it's necessary to declare a whole bunch of *forward* procedures at this point, and to insert some other procedures that will be invoked by *get_x_next*.

⟨ Declarations 8 ⟩ +≡
  static void *mp_scan_primary*(**MP** *mp*);
  static void *mp_scan_secondary*(**MP** *mp*);
  static void *mp_scan_tertiary*(**MP** *mp*);
  static void *mp_scan_expression*(**MP** *mp*);
  static void *mp_scan_suffix*(**MP** *mp*);
  static void *mp_pass_text*(**MP** *mp*);
  static void *mp_conditional*(**MP** *mp*);
  static void *mp_start_input*(**MP** *mp*);
  static void *mp_begin_iteration*(**MP** *mp*);
  static void *mp_resume_iteration*(**MP** *mp*);
  static void *mp_stop_iteration*(**MP** *mp*);

766.   A recursion depth counter is used to discover infinite recursions. (Near) infinite recursion is a problem because it translates into C function calls that eat up the available call stack. A better solution would be to depend on signal trapping, but that is problematic when Metapost is used as a library.

⟨ Global variables 14 ⟩ +≡
  int *expand_depth_count*;   /∗ current expansion depth ∗/
  int *expand_depth*;   /∗ current expansion depth ∗/

767.   The limit is set at 10000, which should be enough to allow normal usages of metapost while preventing the most obvious crashes on most all operating systems, but the value can be raised if the runtime system allows a larger C stack.

⟨ Set initial values of key variables 38 ⟩ +≡
  mp→*expand_depth* = 10000;

768.   Even better would be if the system allows discovery of the amount of space available on the call stack.

In any case, when the limit is crossed, that is a fatal error.

#define   *check_expansion_depth* ( )
>   **if** (++*mp*→*expand_depth_count* ≥ *mp*→*expand_depth*) *mp_expansion_depth_error* (*mp*)

static void *mp_expansion_depth_error* (**MP** *mp*)
{
>   **const char** ∗*hlp*[ ] = {"Recursive␣macro␣expansion␣cannot␣be␣unlimited␣because␣of␣runtime",
>   "stack␣constraints.␣The␣limit␣is␣10000␣recursion␣levels␣in␣total.", Λ};

>   **if** (*mp*→*interaction* ≡ *mp_error_stop_mode*) *mp*→*interaction* = *mp_scroll_mode* ;
>   /∗ no more interaction ∗/
>   **if** (*mp*→*log_opened*) *mp_error* (*mp*, "Maximum␣expansion␣depth␣reached", *hlp*, *true*);
>   *mp*→*history* = *mp_fatal_error_stop* ;
>   *mp_jump_out* (*mp*);
}

769.   An auxiliary subroutine called *expand* is used by *get_x_next* when it has to do exotic expansion commands.

> **static void** *mp_expand* (**MP** *mp*)
> {
>   **size_t** *k*;   /∗ something that we hope is ≤ *buf_size* ∗/
>   **size_t** *j*;   /∗ index into *str_pool* ∗/
>   *check_expansion_depth* ( );
>   **if** (*number_greater* (*internal_value* (*mp_tracing_commands*), *unity_t*))
>   **if** (*cur_cmd* ( ) ≠ *mp_defined_macro*) *show_cur_cmd_mod*;
>   **switch** (*cur_cmd* ( )) {
>   **case** *mp_if_test*: *mp_conditional* (*mp*);   /∗ this procedure is discussed in Part 36 below ∗/
>   **break**;
>   **case** *mp_fi_or_else*: ⟨Terminate the current conditional and skip to **fi** 820⟩;
>   **break**;
>   **case** *mp_input*: ⟨Initiate or terminate input from a file 773⟩;
>   **break**;
>   **case** *mp_iteration*:
>   **if** (*cur_mod* ( ) ≡ *end_for*) {
>   ⟨Scold the user for having an extra **endfor** 770⟩;
>   }
>   **else** {
>   *mp_begin_iteration* (*mp*);   /∗ this procedure is discussed in Part 37 below ∗/
>   }
>   **break**;
>   **case** *mp_repeat_loop*: ⟨Repeat a loop 774⟩;
>   **break**;
>   **case** *mp_exit_test*: ⟨Exit a loop if the proper time has come 775⟩;
>   **break**;
>   **case** *mp_relax*: **break**;
>   **case** *mp_expand_after*: ⟨Expand the token after the next token 777⟩;
>   **break**;
>   **case** *mp_scan_tokens*: ⟨Put a string into the input buffer 778⟩;
>   **break**;
>   **case** *mp_defined_macro*: *mp_macro_call* (*mp*, *cur_mod_node* ( ), Λ, *cur_sym* ( ));
>   **break**;
>   **default**: **break**;   /∗ make the compiler happy ∗/
>   }
>   ;   /∗ there are no other cases ∗/
>   *mp*→*expand_depth_count* −−;
> }

770.   ⟨Scold the user for having an extra **endfor** 770⟩ ≡
> {
>   **const char** ∗*hlp*[ ] = {"I'm␣not␣currently␣working␣on␣a␣for␣loop,",
>   "so␣I␣had␣better␣not␣try␣to␣end␣anything.", Λ};
>   *mp_error* (*mp*, "Extra␣'endfor'", *hlp*, *true*);
>   ;
> }

This code is used in section 769.

771.   The processing of **input** involves the *start_input* subroutine, which will be declared later; the processing of **endinput** is trivial.

⟨ Put each of METAPOST's primitives into the hash table 200 ⟩ +≡
  mp_primitive(*mp*, "input", *mp_input*, 0);
  ;
  mp_primitive(*mp*, "endinput", *mp_input*, 1);

772.   ⟨ Cases of *print_cmd_mod* for symbolic printing of primitives 233 ⟩ +≡
case *mp_input*:
  if (*m* ≡ 0) *mp_print*(*mp*, "input");
  else *mp_print*(*mp*, "endinput");
  break;

773.   ⟨ Initiate or terminate input from a file 773 ⟩ ≡
  if (*cur_mod*( ) > 0) *mp*→*force_eof* = *true*;
  else *mp_start_input*(*mp*)
This code is used in section 769.

774.   We'll discuss the complicated parts of loop operations later. For now it suffices to know that there's a global variable called *loop_ptr* that will be Λ if no loop is in progress.

⟨ Repeat a loop 774 ⟩ ≡
  {
    while (*token_state* ∧ (*nloc* ≡ Λ)) *mp_end_token_list*(*mp*);   /* conserve stack space */
    if (*mp*→*loop_ptr* ≡ Λ) {
      const char **hlp*[ ] = {"I'm␣confused;␣after␣exiting␣from␣a␣loop,␣I␣still␣seem",
        "to␣want␣to␣repeat␣it.␣I'll␣try␣to␣forget␣the␣problem.", Λ};
      mp_error(*mp*, "Lost␣loop", *hlp*, *true*);
      ;
    }
    else {
      mp_resume_iteration(*mp*);   /* this procedure is in Part 37 below */
    }
  }
This code is used in section 769.

775.   ⟨Exit a loop if the proper time has come 775⟩ ≡
  {
    mp_get_boolean(*mp*);
    if (*number_greater*(*internal_value*(*mp_tracing_commands*), *unity_t*))
      mp_show_cmd_mod(*mp*, *mp_nullary*, *cur_exp_value_boolean*());
    if (*cur_exp_value_boolean*() ≡ *mp_true_code*) {
      if (*mp*→*loop_ptr* ≡ Λ) {
        const char *hlp[] = {"Why␣say␣`exitif'␣when␣there's␣nothing␣to␣exit␣from?", Λ};

        if (*cur_cmd*() ≡ *mp_semicolon*) *mp_error*(*mp*, "No␣loop␣is␣in␣progress", *hlp*, *true*);
        else *mp_back_error*(*mp*, "No␣loop␣is␣in␣progress", *hlp*, *true*);
        ;
      }
      else {
        ⟨Exit prematurely from an iteration 776⟩;
      }
    }
    else if (*cur_cmd*() ≠ *mp_semicolon*) {
      const char *hlp[] = {"After␣`exitif␣<boolean␣exp>'␣I␣expect␣to␣see␣a␣semicolon.",
          "I␣shall␣pretend␣that␣one␣was␣there.", Λ};

      mp_back_error(*mp*, "Missing␣`;'␣has␣been␣inserted", *hlp*, *true*);
      ;
    }
  }

This code is used in section 769.

776.   Here we use the fact that *forever_text* is the only *token_type* that is less than *loop_text*.
⟨Exit prematurely from an iteration 776⟩ ≡
  {
    mp_node *p* = Λ;

    do {
      if (*file_state*) {
        mp_end_file_reading(*mp*);
      }
      else {
        if (*token_type* ≤ *loop_text*) *p* = *nstart*;
        mp_end_token_list(*mp*);
      }
    } **while** (*p* ≡ Λ);
    if (*p* ≠ *mp*→*loop_ptr*→*info*) *mp_fatal_error*(*mp*, "***␣(loop␣confusion)");
    ;
    mp_stop_iteration(*mp*);   /* this procedure is in Part 34 below */
  }

This code is used in section 775.

777.   ⟨Expand the token after the next token 777⟩ ≡

  {

    mp_node p;

    $get\_t\_next(mp)$;
    $p = mp\_cur\_tok(mp)$;
    $get\_t\_next(mp)$;
    if $(cur\_cmd(\,) < mp\_min\_command)$ $mp\_expand(mp)$;
    else $mp\_back\_input(mp)$;
    $back\_list(p)$;

  }

This code is used in section 769.

778.   ⟨Put a string into the input buffer 778⟩ ≡

  {

    $mp\_get\_x\_next(mp)$;
    $mp\_scan\_primary(mp)$;
    if $(mp \rightarrow cur\_exp.type \neq mp\_string\_type)$ {
      mp_value new_expr;
      const char $*hlp[\,] = \{$"I'm␣going␣to␣flush␣this␣expression,␣since",
        "scantokens␣should␣be␣followed␣by␣a␣known␣string.", $\Lambda\}$;

      $memset(\&new\_expr, 0, \mathbf{sizeof}(\mathbf{mp\_value}))$;
      $new\_number(new\_expr.data.n)$;
      $mp\_disp\_err(mp, \Lambda)$;
      $mp\_back\_error(mp,$ "Not␣a␣string", $hlp, true)$;
      ;
      $mp\_get\_x\_next(mp)$;
      $mp\_flush\_cur\_exp(mp, new\_expr)$;
    }
    else {
      $mp\_back\_input(mp)$;
      if $(cur\_exp\_str(\,) \rightarrow len > 0)$ ⟨Pretend we're reading a new one-line file 779⟩;
    }

  }

This code is used in section 769.

779.   ⟨Pretend we're reading a new one-line file 779⟩ ≡

```
{
   mp_value new_expr;
   memset(&new_expr, 0, sizeof(mp_value));
   new_number(new_expr.data.n);
   mp_begin_file_reading(mp);
   name = is_scantok;
   k = mp→first + (size_t) cur_exp_str()→len;
   if (k ≥ mp→max_buf_stack) {
      while (k ≥ mp→buf_size) {
         mp_reallocate_buffer(mp, (mp→buf_size + (mp→buf_size/4)));
      }
      mp→max_buf_stack = k + 1;
   }
   j = 0;
   limit = (halfword) k;
   while (mp→first < (size_t) limit) {
      mp→buffer[mp→first] = *(cur_exp_str()→str + j);
      j++;
      incr(mp→first);
   }
   mp→buffer[limit] = xord('%');
   mp→first = (size_t)(limit + 1);
   loc = start;
   mp_flush_cur_exp(mp, new_expr);
}
```

This code is used in section 778.

780.   Here finally is *get_x_next*.

The expression scanning routines to be considered later communicate via the global quantities *cur_type* and *cur_exp*; we must be very careful to save and restore these quantities while macros are being expanded.

⟨Declarations 8⟩ +≡

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

781.   **void** *mp_get_x_next*(**MP** *mp*)

```
{
   mp_node save_exp;      /* a capsule to save cur_type and cur_exp */
   get_t_next(mp);
   if (cur_cmd() < mp_min_command) {
      save_exp = mp_stash_cur_exp(mp);
      do {
         if (cur_cmd() ≡ mp_defined_macro) mp_macro_call(mp, cur_mod_node(), Λ, cur_sym());
         else mp_expand(mp);
         get_t_next(mp);
      } while (cur_cmd() < mp_min_command);
      mp_unstash_cur_exp(mp, save_exp);      /* that restores cur_type and cur_exp */
   }
}
```

782.   Now let's consider the *macro_call* procedure, which is used to start up all user-defined macros. Since the arguments to a macro might be expressions, *macro_call* is recursive.

The first parameter to *macro_call* points to the reference count of the token list that defines the macro. The second parameter contains any arguments that have already been parsed (see below). The third parameter points to the symbolic token that names the macro. If the third parameter is Λ, the macro was defined by **vardef**, so its name can be reconstructed from the prefix and "at" arguments found within the second parameter.

What is this second parameter? It's simply a linked list of symbolic items, whose *info* fields point to the arguments. In other words, if *arg_list* = Λ, no arguments have been scanned yet; otherwise *mp_info*(*arg_list*) points to the first scanned argument, and *mp_link*(*arg_list*) points to the list of further arguments (if any).

Arguments of type **expr** are so-called capsules, which we will discuss later when we concentrate on expressions; they can be recognized easily because their *link* field is **void**. Arguments of type **suffix** and **text** are token lists without reference counts.

783.   After argument scanning is complete, the arguments are moved to the *param_stack*. (They can't be put on that stack any sooner, because the stack is growing and shrinking in unpredictable ways as more arguments are being acquired.) Then the macro body is fed to the scanner; i.e., the replacement text of the macro is placed at the top of the METAPOST's input stack, so that *get_t_next* will proceed to read it next.

⟨ Declarations 8 ⟩ +≡
    static void *mp_macro_call*(**MP** *mp*, **mp_node** *def_ref*, **mp_node** *arg_list*, **mp_sym** *macro_name*);

784.   **void** *mp_macro_call*(**MP** *mp*, **mp_node** *def_ref*, **mp_node** *arg_list*, **mp_sym** *macro_name*)
  {   /* invokes a user-defined control sequence */
    mp_node *r*;   /* current node in the macro's token list */
    mp_node *p*, *q*;   /* for list manipulation */
    integer *n*;   /* the number of arguments */
    mp_node *tail* = 0;   /* tail of the argument list */
    mp_sym *l_delim* = Λ, *r_delim* = Λ;   /* a delimiter pair */
    r = *mp_link*(*def_ref*);
    add_mac_ref(*def_ref*);
    if (*arg_list* ≡ Λ) {
      n = 0;
    }
    else {
      ⟨ Determine the number *n* of arguments already supplied, and set *tail* to the tail of *arg_list* 790 ⟩;
    }
    if (*number_positive*(*internal_value*(*mp_tracing_macros*))) {
      ⟨ Show the text of the macro being expanded, and the existing arguments 785 ⟩;
    }
    ⟨ Scan the remaining arguments, if any; set *r* to the first token of the replacement text 791 ⟩;
    ⟨ Feed the arguments and replacement text to the scanner 803 ⟩;
  }

785.   ⟨Show the text of the macro being expanded, and the existing arguments 785⟩ ≡

$mp\_begin\_diagnostic(mp);$
$mp\_print\_ln(mp);$
$mp\_print\_macro\_name(mp, arg\_list, macro\_name);$
if $(n \equiv 3)$ $mp\_print(mp, \texttt{"@\#"});$ /∗ indicate a suffixed macro ∗/
$mp\_show\_macro(mp, def\_ref, \Lambda, 100000);$
if $(arg\_list \neq \Lambda)$ {
    $n = 0;$
    $p = arg\_list;$
    do {
        $q = (\textbf{mp\_node})\ mp\_sym\_sym(p);$
        $mp\_print\_arg(mp, q, n, 0, 0);$
        $incr(n);$
        $p = mp\_link(p);$
    } **while** $(p \neq \Lambda);$
}
$mp\_end\_diagnostic(mp, false)$

This code is used in section 784.

786.   ⟨Declarations 8⟩ +≡

static void $mp\_print\_macro\_name(\textbf{MP}\ mp, \textbf{mp\_node}\ a, \textbf{mp\_sym}\ n);$

787.   **void** $mp\_print\_macro\_name(\textbf{MP}\ mp, \textbf{mp\_node}\ a, \textbf{mp\_sym}\ n)$
{
    mp_node $p,\ q;$ /∗ they traverse the first part of a ∗/
    if $(n \neq \Lambda)$ {
        $mp\_print\_text(n);$
    }
    else {
        $p = (\textbf{mp\_node})\ mp\_sym\_sym(a);$
        if $(p \equiv \Lambda)$ {
            $mp\_print\_text(mp\_sym\_sym((\textbf{mp\_node})\ mp\_sym\_sym(mp\_link(a))));$
        }
        else {
            $q = p;$
            while $(mp\_link(q) \neq \Lambda)$ $q = mp\_link(q);$
            $mp\_link(q) = (\textbf{mp\_node})\ mp\_sym\_sym(mp\_link(a));$
            $mp\_show\_token\_list(mp, p, \Lambda, 1000, 0);$
            $mp\_link(q) = \Lambda;$
        }
    }
}

788.   ⟨Declarations 8⟩ +≡

static void $mp\_print\_arg(\textbf{MP}\ mp, \textbf{mp\_node}\ q, \textbf{integer}\ n, \textbf{halfword}\ b, \textbf{quarterword}\ bb);$

789.   **void** $mp\_print\_arg(\mathbf{MP}\ mp, \mathbf{mp\_node}\ q, \mathbf{integer}\ n, \mathbf{halfword}\ b, \mathbf{quarterword}\ bb)$
$\{$
   if $(q \wedge mp\_link(q) \equiv \mathtt{MP\_VOID})\ \{$
     $mp\_print\_nl(mp, \texttt{"(EXPR"});$
   $\}$
   else $\{$
     if $((bb < mp\_text\_sym) \wedge (b \neq mp\_text\_macro))\ mp\_print\_nl(mp, \texttt{"(SUFFIX"});$
     else $mp\_print\_nl(mp, \texttt{"(TEXT"});$
   $\}$
   $mp\_print\_int(mp, n);$
   $mp\_print(mp, \texttt{")<-"});$
   if $(q \wedge mp\_link(q) \equiv \mathtt{MP\_VOID})\ mp\_print\_exp(mp, q, 1);$
   else $mp\_show\_token\_list(mp, q, \Lambda, 1000, 0);$
$\}$

790. $\langle$ Determine the number n of arguments already supplied, and set $tail$ to the tail of arg_list 790 $\rangle \equiv$
$\{$
   $n = 1;$
   $tail = arg\_list;$
   while $(mp\_link(tail) \neq \Lambda)\ \{$
     $incr(n);$
     $tail = mp\_link(tail);$
   $\}$
$\}$

This code is used in section 784.

791.   ⟨Scan the remaining arguments, if any; set r to the first token of the replacement text 791⟩ ≡
  $set\_cur\_cmd(mp\_comma + 1)$;   /∗ anything $<>$ $comma$ will do ∗/
  while $(mp\_name\_type(r) \equiv mp\_expr\_sym \lor mp\_name\_type(r) \equiv mp\_suffix\_sym \lor mp\_name\_type(r) \equiv$
        $mp\_text\_sym)$ {
    ⟨Scan the delimited argument represented by $mp\_sym\_info(r)$ 792⟩;
    $r = mp\_link(r)$;
  }
  if $(cur\_cmd() \equiv mp\_comma)$ {
    char $msg[256]$;
    const char $*hlp[\,] = \{$"I'm␣going␣to␣assume␣that␣the␣comma␣I␣just␣read␣was␣a",
        "right␣delimiter,␣and␣then␣I'll␣begin␣expanding␣the␣macro.",
        "You␣might␣want␣to␣delete␣some␣tokens␣before␣continuing.", $\Lambda\}$;
    mp_string $rname$;
    int $old\_setting = mp{\rightarrow}selector$;

    $mp{\rightarrow}selector = new\_string$;
    $mp\_print\_macro\_name(mp, arg\_list, macro\_name)$;
    $rname = mp\_make\_string(mp)$;
    $mp{\rightarrow}selector = old\_setting$;
    $mp\_snprintf(msg, 256,$ "Too␣many␣arguments␣to␣%s;␣Missing␣'%s'␣has␣been␣inserted",
        $mp\_str(mp, rname), mp\_str(mp, text(r\_delim)))$;
    $delete\_str\_ref(rname)$;
    ;
    ;
    $mp\_error(mp, msg, hlp, true)$;
  }
  if $(mp\_sym\_info(r) \neq mp\_general\_macro)$ {
    ⟨Scan undelimited argument(s) 800⟩;
  }
  $r = mp\_link(r)$

This code is used in section 784.

792.   At this point, the reader will find it advisable to review the explanation of token list format that was presented earlier, paying special attention to the conventions that apply only at the beginning of a macro's token list.

On the other hand, the reader will have to take the expression-parsing aspects of the following program on faith; we will explain *cur_type* and *cur_exp* later. (Several things in this program depend on each other, and it's necessary to jump into the circle somewhere.)

⟨ Scan the delimited argument represented by *mp_sym_info*(*r*) 792 ⟩ ≡
  if (*cur_cmd*( ) ≠ *mp_comma*) {
    mp_get_x_next(*mp*);
    if (*cur_cmd*( ) ≠ *mp_left_delimiter*) {
      char *msg*[256];
      const char **hlp*[ ] = {"That␣macro␣has␣more␣parameters␣than␣you␣thought.",
          "I'll␣continue␣by␣pretending␣that␣each␣missing␣argument",
          "is␣either␣zero␣or␣null.", Λ};
      mp_string *sname*;
      int *old_setting* = *mp*→*selector*;

      mp→*selector* = *new_string*;
      mp_print_macro_name(*mp*, *arg_list*, *macro_name*);
      sname = *mp_make_string*(*mp*);
      mp→*selector* = *old_setting*;
      mp_snprintf(*msg*, 256, "Missing␣argument␣to␣%s", *mp_str*(*mp*, *sname*));
      ;
      delete_str_ref(*sname*);
      if (*mp_name_type*(*r*) ≡ *mp_suffix_sym* ∨ *mp_name_type*(*r*) ≡ *mp_text_sym*) {
        set_cur_exp_value_number(*zero_t*);   /* todo: this was *null* */
        mp→*cur_exp.type* = *mp_token_list*;
      }
      else {
        set_cur_exp_value_number(*zero_t*);
        mp→*cur_exp.type* = *mp_known*;
      }
      mp_back_error(*mp*, *msg*, *hlp*, *true*);
      set_cur_cmd((*mp_variable_type*)*mp_right_delimiter*);
      goto FOUND;
    }
    l_delim = *cur_sym*( );
    r_delim = *equiv_sym*(*cur_sym*( ));
  }
  ⟨ Scan the argument represented by *mp_sym_info*(*r*) 795 ⟩;
  if (*cur_cmd*( ) ≠ *mp_comma*) ⟨ Check that the proper right delimiter was present 793 ⟩;
  FOUND: ⟨ Append the current expression to *arg_list* 794 ⟩

This code is used in section 791.

793. ⟨Check that the proper right delimiter was present 793⟩ ≡
  if ((*cur_cmd*( ) ≠ *mp_right_delimiter*) ∨ (*equiv_sym*(*cur_sym*( )) ≠ *l_delim*)) {
    if (*mp_name_type*(*mp_link*(*r*)) ≡ *mp_expr_sym* ∨ *mp_name_type*(*mp_link*(*r*)) ≡
       mp_suffix_sym ∨ *mp_name_type*(*mp_link*(*r*)) ≡ *mp_text_sym*) {
      const char ∗*hlp*[ ] = {"I've␣finished␣reading␣a␣macro␣argument␣and␣am␣about␣to",
        "read␣another;␣the␣arguments␣weren't␣delimited␣correctly.",
        "You␣might␣want␣to␣delete␣some␣tokens␣before␣continuing.", Λ};
      mp_back_error(*mp*, "Missing␣`,'␣has␣been␣inserted", *hlp*, *true*);
      ;
      set_cur_cmd((*mp_variable_type*)*mp_comma*);
    }
    else {
      char *msg*[256];
      const char ∗*hlp*[ ] = {"I've␣gotten␣to␣the␣end␣of␣the␣macro␣parameter␣list.",
        "You␣might␣want␣to␣delete␣some␣tokens␣before␣continuing.", Λ};
      mp_snprintf(*msg*, 256, "Missing␣`%s'␣has␣been␣inserted", *mp_str*(*mp*, *text*(*r_delim*)));
      ;
      mp_back_error(*mp*, *msg*, *hlp*, *true*);
    }
  }

This code is used in section 792.

794. A **suffix** or **text** parameter will have been scanned as a token list pointed to by *cur_exp*, in which case we will have *cur_type* = *token_list*.

⟨Append the current expression to *arg_list* 794⟩ ≡
  {
    p = *mp_get_symbolic_node*(*mp*);
    if (*mp*→*cur_exp.type* ≡ *mp_token_list*) *set_mp_sym_sym*(*p*, *mp*→*cur_exp.data.node*);
    else *set_mp_sym_sym*(*p*, *mp_stash_cur_exp*(*mp*));
    if (*number_positive*(*internal_value*(*mp_tracing_macros*))) {
      mp_begin_diagnostic(*mp*);
      mp_print_arg(*mp*, (**mp_node**) *mp_sym_sym*(*p*), *n*, *mp_sym_info*(*r*), *mp_name_type*(*r*));
      mp_end_diagnostic(*mp*, *false*);
    }
    if (*arg_list* ≡ Λ) {
      arg_list = *p*;
    }
    else {
      mp_link(*tail*) = *p*;
    }
    tail = *p*;
    incr(*n*);
  }

This code is used in sections 792 and 800.

795.   ⟨Scan the argument represented by *mp_sym_info*(*r*) 795⟩ ≡
  if (*mp_name_type*(*r*) ≡ *mp_text_sym*) {
    mp_scan_text_arg(*mp*, *l_delim*, *r_delim*);
  }
  else {
    mp_get_x_next(*mp*);
    if (*mp_name_type*(*r*) ≡ *mp_suffix_sym*) *mp_scan_suffix*(*mp*);
    else *mp_scan_expression*(*mp*);
  }

This code is used in section 792.

796.   The parameters to *scan_text_arg* are either a pair of delimiters or zero; the latter case is for undelimited text arguments, which end with the first semicolon or **endgroup** or **end** that is not contained in a group.

⟨Declarations 8⟩ +≡
  static void *mp_scan_text_arg*(**MP** *mp*, **mp_sym** *l_delim*, **mp_sym** *r_delim*);

797.   **void** *mp_scan_text_arg*(**MP** *mp*, **mp_sym** *l_delim*, **mp_sym** *r_delim*)
  {
    integer *balance*;   /∗ excess of *l_delim* over *r_delim* ∗/
    mp_node *p*;   /∗ list tail ∗/
    mp→warning_info = *l_delim*;
    mp→scanner_status = *absorbing*;
    p = *mp→hold_head*;
    balance = 1;
    mp_link(*mp→hold_head*) = Λ;
    while (1) {
      get_t_next(*mp*);
      if (*l_delim* ≡ Λ) {
        ⟨Adjust the balance for an undelimited argument; **break** if done 799⟩;
      }
      else {
        ⟨Adjust the balance for a delimited argument; **break** if done 798⟩;
      }
      mp_link(*p*) = *mp_cur_tok*(*mp*);
      p = *mp_link*(*p*);
    }
    set_cur_exp_node(*mp_link*(*mp→hold_head*));
    mp→cur_exp.type = *mp_token_list*;
    mp→scanner_status = *normal*;
  }

798.   ⟨ Adjust the balance for a delimited argument; **break** if done 798 ⟩ ≡

 if $(cur\_cmd( ) \equiv mp\_right\_delimiter)$ {
  if $(equiv\_sym(cur\_sym( )) \equiv l\_delim)$ {
   $decr(balance)$;
   if $(balance \equiv 0)$ **break**;
  }
 }
 else if $(cur\_cmd( ) \equiv mp\_left\_delimiter)$ {
  if $(equiv\_sym(cur\_sym( )) \equiv r\_delim)$ $incr(balance)$;
 }

This code is used in section 797.

799.   ⟨ Adjust the balance for an undelimited argument; **break** if done 799 ⟩ ≡

 if $(mp\_end\_of\_statement)$ { /* $cur\_cmd = semicolon$, end_group, or $stop$ */
  if $(balance \equiv 1)$ {
   break;
  }
  else {
   if $(cur\_cmd( ) \equiv mp\_end\_group)$ $decr(balance)$;
  }
 }
 else if $(cur\_cmd( ) \equiv mp\_begin\_group)$ {
  $incr(balance)$;
 }

This code is used in section 797.

800.   ⟨Scan undelimited argument(s) 800⟩ ≡
  {
    if $(mp\_sym\_info(r) < mp\_text\_macro)$ {
       $mp\_get\_x\_next(mp)$;
       if $(mp\_sym\_info(r) \neq mp\_suffix\_macro)$ {
          if $((cur\_cmd() \equiv mp\_equals) \vee (cur\_cmd() \equiv mp\_assignment))$ $mp\_get\_x\_next(mp)$;
       }
    }
    switch $(mp\_sym\_info(r))$ {
    case $mp\_primary\_macro$: $mp\_scan\_primary(mp)$;
       break;
    case $mp\_secondary\_macro$: $mp\_scan\_secondary(mp)$;
       break;
    case $mp\_tertiary\_macro$: $mp\_scan\_tertiary(mp)$;
       break;
    case mp_expr_macro: $mp\_scan\_expression(mp)$;
       break;
    case mp_of_macro: ⟨Scan an expression followed by '**of** ⟨primary⟩' 801⟩;
       break;
    case mp_suffix_macro: ⟨Scan a suffix with optional delimiters 802⟩;
       break;
    case mp_text_macro: $mp\_scan\_text\_arg(mp, \Lambda, \Lambda)$;
       break;
    }   /* there are no other cases */
    $mp\_back\_input(mp)$;
    ⟨Append the current expression to arg_list 794⟩;
  }

This code is used in section 791.

801.   ⟨Scan an expression followed by '**of** ⟨primary⟩' 801⟩ ≡
  {
    $mp\_scan\_expression(mp)$;
    $p = mp\_get\_symbolic\_node(mp)$;
    $set\_mp\_sym\_sym(p, mp\_stash\_cur\_exp(mp))$;
    if $(number\_positive(internal\_value(mp\_tracing\_macros)))$ {
      $mp\_begin\_diagnostic(mp)$;
      $mp\_print\_arg(mp, (\textbf{mp\_node})\ mp\_sym\_sym(p), n, 0, 0)$;
      $mp\_end\_diagnostic(mp, false)$;
    }
    if $(arg\_list \equiv \Lambda)\ arg\_list = p$;
    else $mp\_link(tail) = p$;
    $tail = p$;
    $incr(n)$;
    if $(cur\_cmd() \neq mp\_of\_token)$ {
      char $msg[256]$;
      mp_string $sname$;
      const char $*hlp[\,] = \{$"I've␣got␣the␣first␣argument;␣will␣look␣now␣for␣the␣other.", $\Lambda\}$;
      int $old\_setting = mp{\rightarrow}selector$;

      $mp{\rightarrow}selector = new\_string$;
      $mp\_print\_macro\_name(mp, arg\_list, macro\_name)$;
      $sname = mp\_make\_string(mp)$;
      $mp{\rightarrow}selector = old\_setting$;
      $mp\_snprintf(msg, 256,$ "Missing␣'of'␣has␣been␣inserted␣for␣%s", $mp\_str(mp, sname))$;
      $delete\_str\_ref(sname)$;
      ;
      $mp\_back\_error(mp, msg, hlp, true)$;
    }
    $mp\_get\_x\_next(mp)$;
    $mp\_scan\_primary(mp)$;
  }
This code is used in section 800.

802.   ⟨Scan a suffix with optional delimiters 802⟩ ≡
  {
    if (*cur_cmd*( ) ≠ *mp_left_delimiter*) {
      l_delim = Λ;
    }
    else {
      l_delim = *cur_sym*( );
      r_delim = *equiv_sym*(*cur_sym*( ));
      mp_get_x_next(*mp*);
    }
    mp_scan_suffix(*mp*);
    if (*l_delim* ≠ Λ) {
      if ((*cur_cmd*( ) ≠ *mp_right_delimiter*) ∨ (*equiv_sym*(*cur_sym*( )) ≠ *l_delim*)) {
        char *msg*[256];
        const char *∗hlp*[ ] = {"I've␣gotten␣to␣the␣end␣of␣the␣macro␣parameter␣list.",
            "You␣might␣want␣to␣delete␣some␣tokens␣before␣continuing.", Λ};
        mp_snprintf(*msg*, 256, "Missing␣'%s'␣has␣been␣inserted", *mp_str*(*mp*, *text*(*r_delim*)));
        ;
        mp_back_error(*mp*, *msg*, *hlp*, *true*);
      }
      mp_get_x_next(*mp*);
    }
  }

This code is used in section 800.

803.   Before we put a new token list on the input stack, it is wise to clean off all token lists that have recently been depleted. Then a user macro that ends with a call to itself will not require unbounded stack space.

⟨Feed the arguments and replacement text to the scanner 803⟩ ≡
  while (*token_state* ∧ (*nloc* ≡ Λ)) *mp_end_token_list*(*mp*);   /∗ conserve stack space ∗/
  if (*mp*→*param_ptr* + *n* > *mp*→*max_param_stack*) {
    mp→*max_param_stack* = *mp*→*param_ptr* + *n*;
    mp_check_param_size(*mp*, *mp*→*max_param_stack*);
  }
  mp_begin_token_list(*mp*, *def_ref*, (**quarterword**) *macro*);
  if (*macro_name*) *name* = *text*(*macro_name*);
  else *name* = Λ;
  nloc = *r*;
  if (*n* > 0) {
    p = *arg_list*;
    do {
      mp→*param_stack*[*mp*→*param_ptr*] = (**mp_node**) *mp_sym_sym*(*p*);
      incr(*mp*→*param_ptr*);
      p = *mp_link*(*p*);
    } **while** (*p* ≠ Λ);
    mp_flush_node_list(*mp*, *arg_list*);
  }

This code is used in section 784.

804.   It's sometimes necessary to put a single argument onto *param_stack*. The *stack_argument* subroutine does this.

> **static void** *mp_stack_argument*(**MP** *mp*, **mp_node** *p*)
> {
>   **if** (*mp*→*param_ptr* ≡ *mp*→*max_param_stack*) {
>   *incr*(*mp*→*max_param_stack*);
>   *mp_check_param_size*(*mp*, *mp*→*max_param_stack*);
>   }
>   *mp*→*param_stack*[*mp*→*param_ptr*] = *p*;
>   *incr*(*mp*→*param_ptr*);
> }

805.   Conditional processing.   Let's consider now the way **if** commands are handled.

Conditions can be inside conditions, and this nesting has a stack that is independent of other stacks. Four global variables represent the top of the condition stack: *cond_ptr* points to pushed-down entries, if any; *cur_if* tells whether we are processing **if** or **elseif**; *if_limit* specifies the largest code of a *fi_or_else* command that is syntactically legal; and *if_line* is the line number at which the current conditional began.

If no conditions are currently in progress, the condition stack has the special state *cond_ptr* $= \Lambda$, *if_limit* $=$ *normal*, *cur_if* $= 0$, *if_line* $= 0$. Otherwise *cond_ptr* points to a non-symbolic node; the *type*, *name_type*, and *link* fields of the first word contain *if_limit*, *cur_if*, and *cond_ptr* at the next level, and the second word contains the corresponding *if_line*.

#define   *if_line_field*(A)   ((mp_if_node)(A))→if_line_field_
#define   *if_code*   1   /∗ code for **if** being evaluated ∗/
#define   *fi_code*   2   /∗ code for **fi** ∗/
#define   *else_code*   3   /∗ code for **else** ∗/
#define   *else_if_code*   4   /∗ code for **elseif** ∗/
⟨ MPlib internal header stuff 6 ⟩ +≡
  typedef struct mp_if_node_data {
    NODE_BODY;

    int *if_line_field_*;
  } **mp_if_node_data**;
  typedef struct mp_if_node_data ∗**mp_if_node**;

806.

#define   *if_node_size*   **sizeof**(**struct mp_if_node_data**)
          /∗ number of words in stack entry for conditionals ∗/
  static mp_node *mp_get_if_node*(**MP** *mp*)
  {
    mp_if_node *p* = (**mp_if_node**) *malloc_node*(*if_node_size*);

    mp_type(*p*) = *mp_if_node_type*;
    return (**mp_node**) *p*;
  }

807.   ⟨ Global variables 14 ⟩ +≡
  mp_node *cond_ptr*;   /∗ top of the condition stack ∗/
  integer *if_limit*;   /∗ upper bound on *fi_or_else* codes ∗/
  quarterword *cur_if*;   /∗ type of conditional being worked on ∗/
  integer *if_line*;   /∗ line where that conditional began ∗/

808.   ⟨ Set initial values of key variables 38 ⟩ +≡
  mp→*cond_ptr* = Λ;
  mp→*if_limit* = *normal*;
  mp→*cur_if* = 0;
  mp→*if_line* = 0;

809. ⟨Put each of METAPOST's primitives into the hash table 200⟩ +≡
  $mp\_primitive(mp, \texttt{"if"}, mp\_if\_test, if\_code);$
  ;
  $mp\_primitive(mp, \texttt{"fi"}, mp\_fi\_or\_else, fi\_code);$
  $mp\text{→}frozen\_fi = mp\_frozen\_primitive(mp, \texttt{"fi"}, mp\_fi\_or\_else, fi\_code);$
  ;
  $mp\_primitive(mp, \texttt{"else"}, mp\_fi\_or\_else, else\_code);$
  ;
  $mp\_primitive(mp, \texttt{"elseif"}, mp\_fi\_or\_else, else\_if\_code);$

810. ⟨Cases of $print\_cmd\_mod$ for symbolic printing of primitives 233⟩ +≡
case mp_if_test: **case** $mp\_fi\_or\_else$:
  switch (m) {
  case if_code: $mp\_print(mp, \texttt{"if"});$
    break;
  case fi_code: $mp\_print(mp, \texttt{"fi"});$
    break;
  case $else\_code$: $mp\_print(mp, \texttt{"else"});$
    break;
  default: $mp\_print(mp, \texttt{"elseif"});$
    break;
  }
  break;

811. Here is a procedure that ignores text until coming to an **elseif**, **else**, or **fi** at level zero of **if** . . . **fi** nesting. After it has acted, cur_mod will indicate the token that was found.

METAPOST's smallest two command codes are if_test and fi_or_else; this makes the skipping process a bit simpler.

  void $mp\_pass\_text(\textbf{MP}\ mp)$
  {
    integer $l = 0$;
    $mp\text{→}scanner\_status = skipping;$
    $mp\text{→}warning\_line = mp\_true\_line(mp);$
    while (1) {
      $get\_t\_next(mp);$
      if $(cur\_cmd() \leq mp\_fi\_or\_else)$ {
        if $(cur\_cmd() < mp\_fi\_or\_else)$ {
          $incr(l);$
        }
        else {
          if $(l \equiv 0)$ **break**;
          if $(cur\_mod() \equiv fi\_code)$ $decr(l);$
        }
      }
      else {
        ⟨Decrease the string reference count, if the current token is a string 812⟩;
      }
    }
    $mp\text{→}scanner\_status = normal;$
  }

812.   ⟨Decrease the string reference count, if the current token is a string 812⟩ ≡
   if (*cur_cmd*( ) ≡ *mp_string_token*) {
      delete_str_ref (*cur_mod_str* ( ));
   }

This code is used in sections 127, 811, and 1050.

813.   When we begin to process a new **if**, we set *if_limit*: = *if_code*; then if **elseif** or **else** or **fi** occurs before the current **if** condition has been evaluated, a colon will be inserted. A construction like 'if fi' would otherwise get METAPOST confused.

⟨Push the condition stack 813⟩ ≡
   {
      p = *mp_get_if_node* (*mp*);
      mp_link (*p*) = *mp*→*cond_ptr*;
      mp_type (*p*) = (**quarterword**) *mp*→*if_limit*;
      mp_name_type (*p*) = *mp*→*cur_if* ;
      if_line_field (*p*) = *mp*→*if_line*;
      mp→*cond_ptr* = *p*;
      mp→*if_limit* = *if_code*;
      mp→*if_line* = *mp_true_line* (*mp*);
      mp→*cur_if* = *if_code*;
   }

This code is used in section 817.

814.   ⟨Pop the condition stack 814⟩ ≡
   {
      mp_node *p* = *mp*→*cond_ptr* ;

      mp→*if_line* = *if_line_field* (*p*);
      mp→*cur_if* = *mp_name_type* (*p*);
      mp→*if_limit* = *mp_type* (*p*);
      mp→*cond_ptr* = *mp_link* (*p*);
      mp_free_node (*mp*, *p*, *if_node_size*);
   }

This code is used in sections 817, 818, and 820.

815.   Here's a procedure that changes the *if_limit* code corresponding to a given value of *cond_ptr*.

```
static void mp_change_if_limit(MP mp, quarterword l, mp_node p)
{
  mp_node q;
  if (p ≡ mp→cond_ptr) {
    mp→if_limit = l;     /* that's the easy case */
  }
  else {
    q = mp→cond_ptr;
    while (1) {
      if (q ≡ Λ) mp_confusion(mp, "if");
      ;     /* clang: dereference of null pointer */
      assert(q);
      if (mp_link(q) ≡ p) {
        mp_type(q) = l;
        return;
      }
      q = mp_link(q);
    }
  }
}
```

816.   The user is supposed to put colons into the proper parts of conditional statements. Therefore, METAPOST has to check for their presence.

```
static void mp_check_colon(MP mp)
{
  if (cur_cmd() ≠ mp_colon) {
    const char *hlp[] = {"There␣should've␣been␣a␣colon␣after␣the␣condition.",
        "I␣shall␣pretend␣that␣one␣was␣there.", Λ};
    mp_back_error(mp, "Missing␣`:'␣has␣been␣inserted", hlp, true);
    ;
  }
}
```

817.   A condition is started when the *get_x_next* procedure encounters an *if_test* command; in that case *get_x_next* calls *conditional*, which is a recursive procedure.

```
void mp_conditional(MP mp)
{
  mp_node save_cond_ptr;    /* cond_ptr corresponding to this conditional */
  int new_if_limit;    /* future value of if_limit */
  mp_node p;    /* temporary register */
  ⟨Push the condition stack 813⟩;
  save_cond_ptr = mp→cond_ptr;
RESWITCH: mp_get_boolean(mp);
  new_if_limit = else_if_code;
  if (number_greater(internal_value(mp_tracing_commands), unity_t)) {
    ⟨Display the boolean value of cur_exp 819⟩;
  }
FOUND: mp_check_colon(mp);
  if (cur_exp_value_boolean() ≡ mp_true_code) {
    mp_change_if_limit(mp, (quarterword) new_if_limit, save_cond_ptr);
    return;    /* wait for elseif, else, or fi */
  }
  ;
  ⟨Skip to elseif or else or fi, then goto done 818⟩;
DONE: mp→cur_if = (quarterword) cur_mod();
  mp→if_line = mp_true_line(mp);
  if (cur_mod() ≡ fi_code) {⟨Pop the condition stack 814⟩}
  else if (cur_mod() ≡ else_if_code) {
    goto RESWITCH;
  }
  else {
    set_cur_exp_value_boolean(mp_true_code);
    new_if_limit = fi_code;
    mp_get_x_next(mp);
    goto FOUND;
  }
}
```

818.   In a construction like 'if if true: 0 = 1: *foo* else: *bar* fi', the first else that we come to after learning that the if is false is not the else we're looking for. Hence the following curious logic is needed.

⟨Skip to elseif or else or fi, then goto done 818⟩ ≡

```
while (1) {
  mp_pass_text(mp);
  if (mp→cond_ptr ≡ save_cond_ptr) goto DONE;
  else if (cur_mod() ≡ fi_code) ⟨Pop the condition stack 814⟩;
}
```

This code is used in section 817.

819.   ⟨Display the boolean value of *cur_exp* 819⟩ ≡

 {

  mp_begin_diagnostic(*mp*);

  if (*cur_exp_value_boolean*( ) ≡ *mp_true_code*) *mp_print*(*mp*, "{true}");

  else *mp_print*(*mp*, "{false}");

  mp_end_diagnostic(*mp*, *false*);

 }

This code is used in section 817.

820.   The processing of conditionals is complete except for the following code, which is actually part of *get_x_next*. It comes into play when **elseif**, **else**, or **fi** is scanned.

⟨Terminate the current conditional and skip to **fi** 820⟩ ≡

 if (*cur_mod*( ) > *mp*→*if_limit*) {

  if (*mp*→*if_limit* ≡ *if_code*) { /∗ condition not yet evaluated ∗/

   const char ∗*hlp*[ ] = {"Something␣was␣missing␣here", Λ};

   mp_back_input(*mp*);

   set_cur_sym(*mp*→*frozen_colon*);

   mp_ins_error(*mp*, "Missing␣`:'␣has␣been␣inserted", *hlp*, *true*);

   ;

  }

  else {

   const char ∗*hlp*[ ] = {"I'm␣ignoring␣this;␣it␣doesn't␣match␣any␣if.", Λ};

   if (*cur_mod*( ) ≡ *fi_code*) {

    mp_error(*mp*, "Extra␣fi", *hlp*, *true*);

    ;

   }

   else if (*cur_mod*( ) ≡ *else_code*) {

    mp_error(*mp*, "Extra␣else", *hlp*, *true*);

   }

   else {

    mp_error(*mp*, "Extra␣elseif", *hlp*, *true*);

   }

  }

 }

 else {

  while (*cur_mod*( ) ≠ *fi_code*) *mp_pass_text*(*mp*); /∗ skip to **fi** ∗/

  ⟨Pop the condition stack 814⟩;

 }

This code is used in section 769.

821.   Iterations.   To bring our treatment of *get_x_next* to a close, we need to consider what METAPOST does when it sees **for**, **forsuffixes**, and **forever**.

There's a global variable *loop_ptr* that keeps track of the **for** loops that are currently active. If *loop_ptr* = Λ, no loops are in progress; otherwise *loop_ptr*.*info* points to the iterative text of the current (innermost) loop, and *loop_ptr*.*link* points to the data for any other loops that enclose the current one.

A loop-control node also has two other fields, called *type* and *list*, whose contents depend on the type of loop:

loop_ptr.*type* = Λ means that the link of *loop_ptr*.*list* points to a list of symbolic nodes whose *info* fields point to the remaining argument values of a suffix list and expression list. In this case, an extra field *loop_ptr*.*start_list* is needed to make sure that *resume_operation* skips ahead.

loop_ptr.*type* = `MP_VOID` means that the current loop is '**forever**'.

loop_ptr.*type* = `PROGRESSION_FLAG` means that *loop_ptr*.*value*, *loop_ptr*.*step_size*, and *loop_ptr*.*final_value* contain the data for an arithmetic progression.

loop_ptr.*type* = *p* > `PROGRESSION_FLAG` means that *p* points to an edge header and *loop_ptr*.*list* points into the graphical object list for that edge header.

#define `PROGRESSION_FLAG` (**mp_node**)(2)   /* Λ + 2 */
                    /* *loop_type* value when *loop_list* points to a progression node */

⟨ Types in the outer block 33 ⟩ +≡
  typedef struct mp_loop_data {
    mp_node *info*;   /* iterative text of this loop */
    mp_node *type*;   /* the special type of this loop, or a pointer into mem */
    mp_node *list*;   /* the remaining list elements */
    mp_node *list_start*;   /* head fo the list of elements */
    mp_number *value*;   /* current arithmetic value */
    mp_number *step_size*;   /* arithmetic step size */
    mp_number *final_value*;   /* end arithmetic value */
    struct mp_loop_data *∗link*;   /* the enclosing loop, if any */
  } **mp_loop_data**;

822.   ⟨ Global variables 14 ⟩ +≡
  mp_loop_data *∗loop_ptr*;   /* top of the loop-control-node stack */

823.   ⟨ Set initial values of key variables 38 ⟩ +≡
  mp→loop_ptr = Λ;

824.   If the expressions that define an arithmetic progression in a **for** loop don't have known numeric values, the *bad_for* subroutine screams at the user.

```
static void mp_bad_for (MP mp, const char *s)
{
  char msg[256];
  mp_value new_expr;
  const char *hlp[] = {"When␣you␣say␣'for␣x=a␣step␣b␣until␣c',",
      "the␣initial␣value␣'a'␣and␣the␣step␣size␣'b'",
      "and␣the␣final␣value␣'c'␣must␣have␣known␣numeric␣values.",
      "I'm␣zeroing␣this␣one.␣Proceed,␣with␣fingers␣crossed.", Λ};

  memset (&new_expr, 0, sizeof (mp_value));
  new_number (new_expr.data.n);
  mp_disp_err (mp, Λ);        /* show the bad expression above the message */
  mp_snprintf (msg, 256, "Improper␣%s␣has␣been␣replaced␣by␣0", s);
  ;
  mp_back_error (mp, msg, hlp, true);
  mp_get_x_next (mp);
  mp_flush_cur_exp (mp, new_expr);
}
```

825.   Here's what METAPOST does when **for**, **forsuffixes**, or **forever** has just been scanned. (This code
requires slight familiarity with expression-parsing routines that we have not yet discussed; but it seems to
belong in the present part of the program, even though the original author didn't write it until later. The
reader may wish to come back to it.)

```
void mp_begin_iteration(MP mp)
{
  halfword m;      /* start_for (for) or start_forsuffixes (forsuffixes) */
  mp_sym n;        /* hash address of the current symbol */
  mp_loop_data *s;      /* the new loop-control node */
  mp_subst_list_item *p = Λ;      /* substitution list for scan_toks */
  mp_node q;      /* link manipulation register */
  m = cur_mod();
  n = cur_sym();
  s = xmalloc(1, sizeof(mp_loop_data));
  s→type = s→list = s→info = s→list_start = Λ;
  s→link = Λ;
  new_number(s→value);
  new_number(s→step_size);
  new_number(s→final_value);
  if (m ≡ start_forever) {
    s→type = MP_VOID;
    p = Λ;
    mp_get_x_next(mp);
  }
  else {
    mp_get_symbol(mp);
    p = xmalloc(1, sizeof(mp_subst_list_item));
    p→link = Λ;
    p→info = cur_sym();
    p→info_mod = cur_sym_mod();
    p→value_data = 0;
    if (m ≡ start_for) {
      p→value_mod = mp_expr_sym;
    }
    else {      /* start_forsuffixes */
      p→value_mod = mp_suffix_sym;
    }
    mp_get_x_next(mp);
    if (cur_cmd() ≡ mp_within_token) {
      ⟨Set up a picture iteration 838⟩;
    }
    else {
      ⟨Check for the assignment in a loop header 826⟩;
      ⟨Scan the values to be used in the loop 836⟩;
    }
  }
  ⟨Check for the presence of a colon 827⟩;
  ⟨Scan the loop text and put it on the loop control stack 829⟩;
  mp_resume_iteration(mp);
}
```

826. ⟨ Check for the assignment in a loop header 826 ⟩ ≡
  if $((cur\_cmd() \neq mp\_equals) \wedge (cur\_cmd() \neq mp\_assignment))$ {
    const char $*hlp[] = \{$"The␣next␣thing␣in␣this␣loop␣should␣have␣been␣'='␣or␣':='.",
        "But␣don't␣worry;␣I'll␣pretend␣that␣an␣equals␣sign",
        "was␣present,␣and␣I'll␣look␣for␣the␣values␣next.", $\Lambda\}$;
    $mp\_back\_error(mp,$"Missing␣'='␣has␣been␣inserted"$, hlp, true)$;
    ;
  }

This code is used in section 825.

827. ⟨ Check for the presence of a colon 827 ⟩ ≡
  if $(cur\_cmd() \neq mp\_colon)$ {
    const char $*hlp[] = \{$"The␣next␣thing␣in␣this␣loop␣should␣have␣been␣a␣':'.",
        "So␣I'll␣pretend␣that␣a␣colon␣was␣present;",
        "everything␣from␣here␣to␣'endfor'␣will␣be␣iterated.", $\Lambda\}$;
    $mp\_back\_error(mp,$"Missing␣':'␣has␣been␣inserted"$, hlp, true)$;
    ;
  }

This code is used in section 825.

828. We append a special $mp\text{-}frozen\_repeat\_loop$ token in place of the '**endfor**' at the end of the loop. This will come through METAPOST's scanner at the proper time to cause the loop to be repeated.

  (If the user tries some shenanigan like '**for** ... **let endfor**', he will be foiled by the get_symbol routine, which keeps frozen tokens unchanged. Furthermore the $mp\text{-}frozen\_repeat\_loop$ is an **outer** token, so it won't be lost accidentally.)

829. ⟨ Scan the loop text and put it on the loop control stack 829 ⟩ ≡
  $q = mp\_get\_symbolic\_node(mp)$;
  $set\_mp\_sym\_sym(q, mp\text{-}frozen\_repeat\_loop)$;
  $mp\text{-}scanner\_status = loop\_defining$;
  $mp\text{-}warning\_info = n$;
  $s\text{-}info = mp\_scan\_toks(mp, mp\_iteration, p, q, 0)$;
  $mp\text{-}scanner\_status = normal$;
  $s\text{-}link = mp\text{-}loop\_ptr$; $mp\text{-}loop\_ptr = s$

This code is used in section 825.

830. ⟨ Initialize table entries 182 ⟩ +≡
  $mp\text{-}frozen\_repeat\_loop = mp\_frozen\_primitive(mp,$"␣ENDFOR"$, mp\_repeat\_loop + mp\_outer\_tag, 0)$;

831.   The loop text is inserted into METAPOST's scanning apparatus by the *resume_iteration* routine.

```
void mp_resume_iteration(MP mp)
{
  mp_node p, q;      /* link registers */
  p = mp→loop_ptr→type;
  if (p ≡ PROGRESSION_FLAG) {
    set_cur_exp_value_number(mp→loop_ptr→value);
    if (⟨The arithmetic progression has ended 832⟩) {
      mp_stop_iteration(mp);
      return;
    }
    mp→cur_exp.type = mp_known;
    q = mp_stash_cur_exp(mp);      /* make q an expr argument */
    set_number_from_addition(mp→loop_ptr→value, cur_exp_value_number( ), mp→loop_ptr→step_size);
      /* set value(p) for the next iteration */      /* detect numeric overflow */
    if (number_positive(mp→loop_ptr→step_size)∧number_less(mp→loop_ptr→value, cur_exp_value_number( )))
      {
      if (number_positive(mp→loop_ptr→final_value)) {
        number_clone(mp→loop_ptr→value, mp→loop_ptr→final_value);
        number_add_scaled(mp→loop_ptr→final_value, −1);
      }
      else {
        number_clone(mp→loop_ptr→value, mp→loop_ptr→final_value);
        number_add_scaled(mp→loop_ptr→value, 1);
      }
    }
    else if (number_negative(mp→loop_ptr→step_size) ∧ number_greater(mp→loop_ptr→value,
          cur_exp_value_number( ))) {
      if (number_negative(mp→loop_ptr→final_value)) {
        number_clone(mp→loop_ptr→value, mp→loop_ptr→final_value);
        number_add_scaled(mp→loop_ptr→final_value, 1);
      }
      else {
        number_clone(mp→loop_ptr→value, mp→loop_ptr→final_value);
        number_add_scaled(mp→loop_ptr→value, −1);
      }
    }
  }
  else if (p ≡ Λ) {
    p = mp→loop_ptr→list;
    if (p ≠ Λ ∧ p ≡ mp→loop_ptr→list_start) {
      q = p;
      p = mp_link(p);
      mp_free_symbolic_node(mp, q);
      mp→loop_ptr→list = p;
    }
    if (p ≡ Λ) {
      mp_stop_iteration(mp);
      return;
    }
    mp→loop_ptr→list = mp_link(p);
    q = (mp_node) mp_sym_sym(p);
```

```
      mp_free_symbolic_node(mp, p);
    }
    else if (p ≡ MP_VOID) {
      mp_begin_token_list(mp, mp→loop_ptr→info, (quarterword) forever_text);
      return;
    }
    else {
      ⟨Make q a capsule containing the next picture component from loop_list(loop_ptr) or goto
          not_found 834⟩;
    }
    mp_begin_token_list(mp, mp→loop_ptr→info, (quarterword) loop_text);
    mp_stack_argument(mp, q);
    if (number_greater(internal_value(mp_tracing_commands), unity_t)) {
      ⟨Trace the start of a loop 833⟩;
    }
    return;
  NOT_FOUND: mp_stop_iteration(mp);
  }
```

832. ⟨The arithmetic progression has ended 832⟩ ≡
  (number_positive(mp→loop_ptr→step_size) ∧ number_greater(cur_exp_value_number( ),
      mp→loop_ptr→final_value)) ∨ (number_negative(mp→loop_ptr→step_size) ∧
      number_less(cur_exp_value_number( ), mp→loop_ptr→final_value))
This code is used in section 831.

833. ⟨Trace the start of a loop 833⟩ ≡
  {
    mp_begin_diagnostic(mp);
    mp_print_nl(mp, "{loop⎵value=");
    ;
    if ((q ≠ Λ) ∧ (mp_link(q) ≡ MP_VOID)) mp_print_exp(mp, q, 1);
    else mp_show_token_list(mp, q, Λ, 50, 0);
    mp_print_char(mp, xord('}'));
    mp_end_diagnostic(mp, false);
  }
This code is used in section 831.

834. ⟨Make q a capsule containing the next picture component from loop_list(loop_ptr) or goto
      not_found 834⟩ ≡
  {
    q = mp→loop_ptr→list;
    if (q ≡ Λ) goto NOT_FOUND;
    if (¬is_start_or_stop(q)) q = mp_link(q);
    else if (¬is_stop(q)) q = mp_skip_1component(mp, q);
    else goto NOT_FOUND;
    set_cur_exp_node((mp_node) mp_copy_objects(mp, mp→loop_ptr→list, q));
    mp_init_bbox(mp, (mp_edge_header_node) cur_exp_node( ));
    mp→cur_exp.type = mp_picture_type;
    mp→loop_ptr→list = q;
    q = mp_stash_cur_exp(mp);
  }
This code is used in section 831.
```

**835.**    A level of loop control disappears when *resume_iteration* has decided not to resume, or when an
**exitif** construction has removed the loop text from the input stack.

```
void mp_stop_iteration(MP mp)
{
  mp_node p, q;      /* the usual */
  mp_loop_data *tmp;      /* for free() */
  p = mp→loop_ptr→type;
  if (p ≡ PROGRESSION_FLAG) {
    mp_free_symbolic_node(mp, mp→loop_ptr→list);
  }
  else if (p ≡ Λ) {
    q = mp→loop_ptr→list;
    while (q ≠ Λ) {
      p = (mp_node) mp_sym_sym(q);
      if (p ≠ Λ) {
        if (mp_link(p) ≡ MP_VOID) {      /* it's an expr parameter */
          mp_recycle_value(mp, p);
          mp_free_value_node(mp, p);
        }
        else {
          mp_flush_token_list(mp, p);      /* it's a suffix or text parameter */
        }
      }
      p = q;
      q = mp_link(q);
      mp_free_symbolic_node(mp, p);
    }
  }
  else if (p > PROGRESSION_FLAG) {
    delete_edge_ref(p);
  }
  tmp = mp→loop_ptr;
  mp→loop_ptr = tmp→link;
  mp_flush_token_list(mp, tmp→info);
  free_number(tmp→value);
  free_number(tmp→step_size);
  free_number(tmp→final_value);
  xfree(tmp);
}
```

836.   Now that we know all about loop control, we can finish up the missing portion of *begin_iteration* and we'll be done.

The following code is performed after the '=' has been scanned in a **for** construction (if $m = start\_for$) or a **forsuffixes** construction (if $m = start\_forsuffixes$).

⟨Scan the values to be used in the loop 836⟩ ≡
```
  s→type = Λ;
  s→list = mp_get_symbolic_node(mp);
  s→list_start = s→list;
  q = s→list; do
  {
    mp_get_x_next(mp);
    if (m ≠ start_for) {
      mp_scan_suffix(mp);
    }
    else {
      if (cur_cmd( ) ≥ mp_colon)
        if (cur_cmd( ) ≤ mp_comma) goto CONTINUE;
      mp_scan_expression(mp);
      if (cur_cmd( ) ≡ mp_step_token)
        if (q ≡ s→list) {
          ⟨Prepare for step-until construction and break 837⟩;
        }
      set_cur_exp_node(mp_stash_cur_exp(mp));
    }
    mp_link(q) = mp_get_symbolic_node(mp);
    q = mp_link(q);
    set_mp_sym_sym(q, mp→cur_exp.data.node);
    if (m ≡ start_for)  mp_name_type(q) = mp_expr_sym;
    else if (m ≡ start_forsuffixes)  mp_name_type(q) = mp_suffix_sym;
    mp→cur_exp.type = mp_vacuous;
  CONTINUE: ;
  }
  while (cur_cmd( ) ≡ mp_comma)
```
This code is used in section 825.

837.   ⟨Prepare for step-until construction and **break** 837⟩ ≡
  {
    if (*mp→cur_exp.type* ≠ *mp_known*) *mp_bad_for*(*mp*, "initial␣value");
    number_clone(*s→value*, *cur_exp_value_number*( ));
    mp_get_x_next(*mp*);
    mp_scan_expression(*mp*);
    if (*mp→cur_exp.type* ≠ *mp_known*) *mp_bad_for*(*mp*, "step␣size");
    number_clone(*s→step_size*, *cur_exp_value_number*( ));
    if (*cur_cmd*( ) ≠ *mp_until_token*) {
      const char ∗*hlp*[ ] = {"I␣assume␣you␣meant␣to␣say␣'until'␣after␣'step'.",
          "So␣I'll␣look␣for␣the␣final␣value␣and␣colon␣next.", Λ};

      mp_back_error(*mp*, "Missing␣'until'␣has␣been␣inserted", *hlp*, *true*);

      ;
    }
    mp_get_x_next(*mp*);
    mp_scan_expression(*mp*);
    if (*mp→cur_exp.type* ≠ *mp_known*) *mp_bad_for*(*mp*, "final␣value");
    number_clone(*s→final_value*, *cur_exp_value_number*( ));
    s→type = PROGRESSION_FLAG;
    break;
  }
This code is used in section 836.

838.   The last case is when we have just seen "**within**", and we need to parse a picture expression and prepare to iterate over it.

⟨Set up a picture iteration 838⟩ ≡
  {
    mp_get_x_next(*mp*);
    mp_scan_expression(*mp*);
    ⟨Make sure the current expression is a known picture 839⟩;
    s→type = *mp→cur_exp.data.node*;
    mp→cur_exp.type = *mp_vacuous*;
    q = *mp_link*(*edge_list*(*mp→cur_exp.data.node*));
    if (*q* ≠ Λ)
      if (*is_start_or_stop*(*q*))
        if (*mp_skip_1component*(*mp*, *q*) ≡ Λ) *q* = *mp_link*(*q*);
    s→list = *q*;
  }
This code is used in section 825.

839.   ⟨ Make sure the current expression is a known picture 839 ⟩ ≡

  if (mp→$cur\_exp.type \neq mp\_picture\_type$) {

    mp_value new_expr;

    const char $*hlp[\,] = \{$"When␣you␣say␣'for␣x␣in␣p',␣p␣must␣be␣a␣known␣picture.", $\Lambda\}$;

    $memset(\&new\_expr, 0, \mathbf{sizeof}(\mathbf{mp\_value}))$;

    $new\_number(new\_expr.data.n)$;

    $new\_expr.data.node = (\mathbf{mp\_node})\ mp\_get\_edge\_header\_node(mp)$;

    $mp\_disp\_err(mp, \Lambda)$;

    $mp\_back\_error(mp,$ "Improper␣iteration␣spec␣has␣been␣replaced␣by␣nullpicture", $hlp, true)$;

    $mp\_get\_x\_next(mp)$;

    $mp\_flush\_cur\_exp(mp, new\_expr)$;

    $mp\_init\_edges(mp, (\mathbf{mp\_edge\_header\_node})\ mp$→$cur\_exp.data.node)$;

    mp→$cur\_exp.type = mp\_picture\_type$;

  }

This code is used in section 838.

840.   File names.   It's time now to fret about file names. Besides the fact that different operating systems treat files in different ways, we must cope with the fact that completely different naming conventions are used by different groups of people. The following programs show what is required for one particular operating system; similar routines for other systems are not difficult to devise.

METAPOST assumes that a file name has three parts: the name proper; its "extension"; and a "file area" where it is found in an external file system. The extension of an input file is assumed to be '.mp' unless otherwise specified; it is '.log' on the transcript file that records each run of METAPOST; it is '.tfm' on the font metric files that describe characters in any fonts created by METAPOST; it is '.ps' or '.nnn' for some number nnn on the PostScript output files. The file area can be arbitrary on input files, but files are usually output to the user's current area. If an input file cannot be found on the specified area, METAPOST will look for it on a special system area; this special area is intended for commonly used input files.

Simple uses of METAPOST refer only to file names that have no explicit extension or area. For example, a person usually says 'input cmr10' instead of 'input cmr10.new'. Simple file names are best, because they make the METAPOST source files portable; whenever a file name consists entirely of letters and digits, it should be treated in the same way by all implementations of METAPOST. However, users need the ability to refer to other files in their environment, especially when responding to error messages concerning unopenable files; therefore we want to let them use the syntax that appears in their favorite operating system.

841.   METAPOST uses the same conventions that have proved to be satisfactory for TEX and METAFONT. In order to isolate the system-dependent aspects of file names, the system-independent parts of META-POST are expressed in terms of three system-dependent procedures called *begin_name*, *more_name*, and *end_name*. In essence, if the user-specified characters of the file name are $c_1 \ldots c_n$, the system-independent driver program does the operations

$$begin\_name;\ more\_name(c_1);\ \ldots;\ more\_name(c_n);\ end\_name.$$

These three procedures communicate with each other via global variables. Afterwards the file name will appear in the string pool as three strings called *cur_name*, *cur_area*, and *cur_ext*; the latter two are NULL (i.e., ""), unless they were explicitly specified by the user.

Actually the situation is slightly more complicated, because METAPOST needs to know when the file name ends. The *more_name* routine is a function (with side effects) that returns *true* on the calls $more\_name(c_1)$, $\ldots$, $more\_name(c_{n-1})$. The final call $more\_name(c_n)$ returns *false*; or, it returns *true* and c_n is the last character on the current input line. In other words, *more_name* is supposed to return *true* unless it is sure that the file name has been completely scanned; and *end_name* is supposed to be able to finish the assembly of *cur_name*, *cur_area*, and *cur_ext* regardless of whether $more\_name(c_n)$ returned *true* or *false*.

⟨ Global variables 14 ⟩ +≡
    char *cur_name;   /∗ name of file just scanned ∗/
    char *cur_area;   /∗ file area just scanned, or "" ∗/
    char *cur_ext;   /∗ file extension just scanned, or "" ∗/

842.   It is easier to maintain reference counts if we assign initial values.

⟨ Set initial values of key variables 38 ⟩ +≡
    $mp\to cur\_name = xstrdup("")$;
    $mp\to cur\_area = xstrdup("")$;
    $mp\to cur\_ext = xstrdup("")$;

843.   ⟨ Dealloc variables 27 ⟩ +≡
    $xfree(mp\to cur\_area)$;
    $xfree(mp\to cur\_name)$;
    $xfree(mp\to cur\_ext)$;

844.   The file names we shall deal with for illustrative purposes have the following structure: If the name contains '`>`' or '`:`', the file area consists of all characters up to and including the final such character; otherwise the file area is null. If the remaining file name contains '`.`', the file extension consists of all such characters from the first remaining '`.`' to the end, otherwise the file extension is null.

We can scan such file names easily by using two global variables that keep track of the occurrences of area and extension delimiters.

⟨ Global variables 14 ⟩ +≡
   integer *area_delimiter*;   /∗ most recent '`>`' or '`:`' relative to *str_start*[*str_ptr*] ∗/
   integer *ext_delimiter*;   /∗ the relevant '`.`', if any ∗/
   boolean *quoted_filename*;   /∗ whether the filename is wrapped in " markers ∗/

845.   Here now is the first of the system-dependent routines for file name scanning.

⟨ Declarations 8 ⟩ +≡
   static void *mp_begin_name*(**MP** *mp*);
   static boolean *mp_more_name*(**MP** *mp*, **ASCII_code** *c*);
   static void *mp_end_name*(**MP** *mp*);

846.   **void** *mp_begin_name*(**MP** *mp*)
  {
     xfree(*mp*→*cur_name*);
     xfree(*mp*→*cur_area*);
     xfree(*mp*→*cur_ext*);
     mp→*area_delimiter* = −1;
     mp→*ext_delimiter* = −1;
     mp→*quoted_filename* = *false*;
  }

847.   And here's the second.

#ifndef `IS_DIR_SEP`
#define `IS_DIR_SEP`(*c*) (*c* ≡ '`/`' ∨ *c* ≡ '`\\`')
#endif
   boolean *mp_more_name*(**MP** *mp*, **ASCII_code** *c*)
  {
    if (*c* ≡ '`"`') {
      mp→*quoted_filename* = ¬*mp*→*quoted_filename*;
    }
    else if ((*c* ≡ '` `' ∨ *c* ≡ '`\t`') ∧ (*mp*→*quoted_filename* ≡ *false*)) {
      return *false*;
    }
    else {
      if (`IS_DIR_SEP`(*c*)) {
        mp→*area_delimiter* = (**integer**) *mp*→*cur_length*;
        mp→*ext_delimiter* = −1;
      }
      else if (*c* ≡ '`.`') {
        mp→*ext_delimiter* = (**integer**) *mp*→*cur_length*;
      }
      append_char(*c*);   /∗ contribute *c* to the current string ∗/
    }
    return *true*;
  }

848.   The third.

#define $copy\_pool\_segment(A, B, C)$
$$\{$$
$$A = xmalloc(C + 1, \textbf{sizeof}(\textbf{char}));$$
$$(\textbf{void})\ memcpy(A, (\textbf{char} *)(mp\text{-}cur\_string + B), C);$$
$$A[C] = 0;$$
$$\}$$

void $mp\_end\_name(\textbf{MP}\ mp)$
{
  size_t $s = 0$;   /* length of area, name, and extension */
  size_t len;   /* "my/w.mp" */
  if $(mp\text{-}area\_delimiter < 0)$ {
    $mp\text{-}cur\_area = xstrdup("");$
  }
  else {
    $len = (\textbf{size\_t})\ mp\text{-}area\_delimiter - s + 1;$
    $copy\_pool\_segment(mp\text{-}cur\_area, s, len);$
    $s += len;$
  }
  if $(mp\text{-}ext\_delimiter < 0)$ {
    $mp\text{-}cur\_ext = xstrdup("");$
    $len = (\textbf{unsigned})(mp\text{-}cur\_length - s);$
  }
  else {
    $copy\_pool\_segment(mp\text{-}cur\_ext, mp\text{-}ext\_delimiter, (mp\text{-}cur\_length - (\textbf{size\_t})\ mp\text{-}ext\_delimiter));$
    $len = (\textbf{size\_t})\ mp\text{-}ext\_delimiter - s;$
  }
  $copy\_pool\_segment(mp\text{-}cur\_name, s, len);$
  $mp\_reset\_cur\_string(mp);$
}

849.   Conversely, here is a routine that takes three strings and prints a file name that might have produced them. (The routine is system dependent, because some operating systems put the file area last instead of first.)

⟨ Basic printing procedures 85 ⟩ +≡
  static void $mp\_print\_file\_name(\textbf{MP}\ mp, \textbf{char}\ *n, \textbf{char}\ *a, \textbf{char}\ *e)$
  {
    boolean $must\_quote = false;$
    if $(((a \neq \Lambda) \wedge (strchr(a, '\sqcup') \neq \Lambda)) \vee ((n \neq \Lambda) \wedge (strchr(n, '\sqcup') \neq \Lambda)) \vee ((e \neq \Lambda) \wedge (strchr(e, '\sqcup') \neq \Lambda)))$
      $must\_quote = true;$
    if $(must\_quote)\ mp\_print\_char(mp, (\textbf{ASCII\_code})\ '"');$
    $mp\_print(mp, a);$
    $mp\_print(mp, n);$
    $mp\_print(mp, e);$
    if $(must\_quote)\ mp\_print\_char(mp, (\textbf{ASCII\_code})\ '"');$
  }

850.   Another system-dependent routine is needed to convert three internal METAPOST strings to the *name_of_file* value that is used to open files. The present code allows both lowercase and uppercase letters in the file name.

#define *append_to_name*(A)
  {
   mp→*name_of_file*[k++] = (**char**) *xchr*(*xord*((**ASCII_code**)(A)));
  }

851.   **void** *mp_pack_file_name*(**MP** *mp*, **const char** ∗*n*, **const char** ∗*a*, **const char** ∗*e*)
 {
  integer *k*; /∗ number of positions filled in *name_of_file* ∗/
  const char ∗*j*; /∗ a character index ∗/
  size_t *slen*;

  k = 0;
  assert(*n* ≠ Λ);
  xfree(*mp*→*name_of_file*);
  slen = *strlen*(*n*) + 1;
  if (*a* ≠ Λ) *slen* += *strlen*(*a*);
  if (*e* ≠ Λ) *slen* += *strlen*(*e*);
  mp→*name_of_file* = *xmalloc*(*slen*, 1);
  if (*a* ≠ Λ) {
   for (*j* = *a*; ∗*j* ≠ '\0'; *j*++) {
    append_to_name(∗*j*);
   }
  }
  for (*j* = *n*; ∗*j* ≠ '\0'; *j*++) {
   append_to_name(∗*j*);
  }
  if (*e* ≠ Λ) {
   for (*j* = *e*; ∗*j* ≠ '\0'; *j*++) {
    append_to_name(∗*j*);
   }
  }
  mp→*name_of_file*[*k*] = 0;
 }

852.   ⟨Internal library declarations 10⟩ +≡
 void *mp_pack_file_name*(**MP** *mp*, **const char** ∗*n*, **const char** ∗*a*, **const char** ∗*e*);

853.   ⟨Option variables 26⟩ +≡
 char ∗*mem_name*; /∗ for commandline ∗/

854.   Stripping a . *mem* extension here is for backward compatibility.

⟨ Find and load preload file, if required 854 ⟩ ≡
  if (¬*opt→ini_version*) {
    mp→mem_name = *xstrdup*(*opt→mem_name*);
    if (*mp→mem_name*) {
      size_t *l* = *strlen*(*mp→mem_name*);
      if (*l* > 4) {
        char *∗test* = *strstr*(*mp→mem_name*, ".mem");
        if (*test* ≡ *mp→mem_name* + *l* − 4) {
          ∗test = 0;
        }
      }
    }
    if (*mp→mem_name* ≠ Λ) {
      if (¬*mp_open_mem_file*(*mp*)) {
        mp→history = *mp_fatal_error_stop*;
        mp_jump_out(*mp*);
      }
    }
  }

This code is used in section 16.

855.   ⟨ Dealloc variables 27 ⟩ +≡
  xfree(*mp→mem_name*);

856.   This part of the program becomes active when a "virgin" METAPOST is trying to get going, just after the preliminary initialization. The buffer contains the first line of input in *buffer*[*loc* .. (*last* − 1)], where *loc* < *last* and *buffer*[*loc*] <> "".

⟨ Declarations 8 ⟩ +≡
  static boolean *mp_open_mem_name*(**MP** *mp*);
  static boolean *mp_open_mem_file*(**MP** *mp*);

857.   **boolean** *mp_open_mem_name*(**MP** *mp*)
{
    if (*mp→mem_name* ≠ Λ) {
        size_t *l* = *strlen*(*mp→mem_name*);
        char ∗*s* = *xstrdup*(*mp→mem_name*);
        if (*l* > 4) {
            char ∗*test* = *strstr*(*s*, ".mp");
            if (*test* ≡ Λ ∨ *test* ≠ *s* + *l* − 4) {
                s = *xrealloc*(*s*, *l* + 5, 1);
                strcat(*s*, ".mp");
            }
        }
        else {
            s = *xrealloc*(*s*, *l* + 5, 1);
            strcat(*s*, ".mp");
        }
        s = (*mp→find_file*)(*mp*, *s*, "r", *mp_filetype_program*);
        xfree(*mp→name_of_file*);
        if (*s* ≡ Λ) **return** *false*;
        mp→name_of_file = *xstrdup*(*s*);
        mp→mem_file = (*mp→open_file*)(*mp*, *s*, "r", *mp_filetype_program*);
        free(*s*);
        if (*mp→mem_file*) **return** *true*;
    }
    return *false*;
}
boolean *mp_open_mem_file*(**MP** *mp*)
{
    if (*mp→mem_file* ≠ Λ) **return** *true*;
    if (*mp_open_mem_name*(*mp*)) **return** *true*;
    if (*mp_xstrcmp*(*mp→mem_name*, "plain")) {
        wake_up_terminal( );
        wterm("Sorry,␣I␣can\'t␣find␣the␣'");
        wterm(*mp→mem_name*);
        wterm("'␣preload␣file;␣will␣try␣'plain'.");
        wterm_cr;
        ;
        update_terminal( );   /∗ now pull out all the stops: try for the system **plain** file ∗/
        xfree(*mp→mem_name*);
        mp→mem_name = *xstrdup*("plain");
        if (*mp_open_mem_name*(*mp*)) **return** *true*;
    }
    wake_up_terminal( );
    wterm_ln("I␣can't␣find␣the␣'plain'␣preload␣file!\n");
    ;
    return *false*;
}

858.   Operating systems often make it possible to determine the exact name (and possible version number) of a file that has been opened. The following routine, which simply makes a METAPOST string from the value of *name_of_file*, should ideally be changed to deduce the full name of file f, which is the file most recently opened, if it is possible to do this.

859.   **static mp_string** *mp_make_name_string*(**MP** *mp*)
{
   int *k*;   /∗ index into *name_of_file* ∗/
   int *name_length* = (**int**) *strlen*(*mp⃗name_of_file*);
   str_room(*name_length*);
   for (*k* = 0; *k* < *name_length*; *k*++) {
     append_char(*xord*((**ASCII_code**) *mp⃗name_of_file*[*k*]));
   }
   return *mp_make_string*(*mp*);
}

860.   Now let's consider the "driver" routines by which METAPOST deals with file names in a system-independent manner. First comes a procedure that looks for a file name in the input by taking the information from the input buffer. (We can't use *get_next*, because the conversion to tokens would destroy necessary information.)

This procedure doesn't allow semicolons or percent signs to be part of file names, because of other conventions of METAPOST. *The METAFONT book* doesn't use semicolons or percents immediately after file names, but some users no doubt will find it natural to do so; therefore system-dependent changes to allow such characters in file names should probably be made with reluctance, and only when an entire file name that includes special characters is "quoted" somehow.

   static void *mp_scan_file_name*(**MP** *mp*)
   {
     mp_begin_name(*mp*);
     while (*mp⃗buffer*[*loc*] ≡ '␣') *incr*(*loc*);
     while (1) {
       if ((*mp⃗buffer*[*loc*] ≡ ';') ∨ (*mp⃗buffer*[*loc*] ≡ '%')) **break**;
       if (¬*mp_more_name*(*mp*, *mp⃗buffer*[*loc*])) **break**;
       incr(*loc*);
     }
     mp_end_name(*mp*);
   }

861.   Here is another version that takes its input from a string.

⟨ Declare subroutines for parsing file names 861 ⟩ ≡
   void *mp_str_scan_file*(**MP** *mp*, **mp_string** *s*);

See also section 863.

This code is used in section 10.

862.   **void** *mp_str_scan_file*(**MP** *mp*, **mp_string** *s*)
{
    size_t *p*, *q*;   /* current position and stopping point */
    mp_begin_name(*mp*);
    p = 0;
    q = *s*→*len*;
    while (*p* < *q*) {
      if (¬*mp_more_name*(*mp*, *(*s*→*str* + *p*))) **break**;
      incr(*p*);
    }
    mp_end_name(*mp*);
}

863.   And one that reads from a **char** *.
⟨Declare subroutines for parsing file names 861⟩ +≡
  extern void *mp_ptr_scan_file*(**MP** *mp*, **char** *s);

864.   **void** *mp_ptr_scan_file*(**MP** *mp*, **char** *s)
{
    char *p, *q;   /* current position and stopping point */
    mp_begin_name(*mp*);
    p = *s*;
    q = *p* + *strlen*(*s*);
    while (*p* < *q*) {
      if (¬*mp_more_name*(*mp*, (**ASCII_code**)(*p))) **break**;
      p++;
    }
    mp_end_name(*mp*);
}

865.   The option variable *job_name* contains the file name that was first **input** by the user. This name is used to initialize the *job_name* global as well as the *mp_job_name* internal, and is extended by '.log' and 'ps' and '.mem' and '.tfm' in order to make the names of METAPOST's output files.

⟨Global variables 14⟩ +≡
  boolean *log_opened*;   /* has the transcript file been opened? */
  char *log_name;   /* full name of the log file */

866.   ⟨Option variables 26⟩ +≡
  char *job_name;   /* principal file name */

867.   Initially *job_name* = Λ; it becomes nonzero as soon as the true name is known. We have *job_name* = Λ if and only if the 'log' file has not been opened, except of course for a short time just after *job_name* has become nonzero.

⟨Allocate or initialize variables 28⟩ +≡
  mp→*job_name* = *mp_xstrdup*(*mp*, *opt*→*job_name*);
    /* **if** (*mp*→*job_name* ≠ Λ) { **char** *s = *mp*→*job_name* + *strlen*(*mp*→*job_name*); **while**
    (*s* > *mp*→*job_name*) { **if** (*s ≡ '.') { *s = '\0'; } *s*−−; } } */
  if (*opt*→*noninteractive*) {
    if (*mp*→*job_name* ≡ Λ) *mp*→*job_name* = *mp_xstrdup*(*mp*, *mp*→*mem_name*);
  }
  mp→*log_opened* = *false*;

868.   Cannot do this earlier because at the $<$ *Allocate* $\lor \dots >$, the string pool is not yet initialized.

$\langle$ Fix up $mp\text{-}internal[mp\_job\_name]$ 868 $\rangle \equiv$
  if $(mp\text{-}job\_name \neq \Lambda)$ {
    if $(internal\_string(mp\_job\_name) \neq 0)$ $delete\_str\_ref(internal\_string(mp\_job\_name))$;
    $set\_internal\_string(mp\_job\_name, mp\_rts(mp, mp\text{-}job\_name))$;
  }

This code is used in sections 16, 875, 880, 1066, and 1251.

869. $\langle$ Dealloc variables 27 $\rangle$ +$\equiv$
  $xfree(mp\text{-}job\_name)$;

870.   Here is a routine that manufactures the output file names, assuming that $job\_name <> 0$. It ignores
and changes the current settings of cur_area and cur_ext.

#define $pack\_cur\_name$ $mp\_pack\_file\_name(mp, mp\text{-}cur\_name, mp\text{-}cur\_area, mp\text{-}cur\_ext)$

$\langle$ Internal library declarations 10 $\rangle$ +$\equiv$
  void $mp\_pack\_job\_name(\mathbf{MP}\ mp, \mathbf{const\ char}\ *s)$;

871.   **void** $mp\_pack\_job\_name(\mathbf{MP}\ mp, \mathbf{const\ char}\ *s)$
  {   /* $s =$ ".log", ".mem", ".ps", or .*nnn* */
  $xfree(mp\text{-}cur\_name)$;
  $mp\text{-}cur\_name = xstrdup(mp\text{-}job\_name)$;
  $xfree(mp\text{-}cur\_area)$;
  $mp\text{-}cur\_area = xstrdup("")$;
  $xfree(mp\text{-}cur\_ext)$;
  $mp\text{-}cur\_ext = xstrdup(s)$;
  $pack\_cur\_name$;
  }

872.   If some trouble arises when METAPOST tries to open a file, the following routine calls upon the user
to supply another file name. Parameter s is used in the error message to identify the type of file; parameter e
is the default extension if none is given. Upon exit from the routine, variables cur_name, cur_area, cur_ext,
and $name\_of\_file$ are ready for another attempt at file opening.

$\langle$ Internal library declarations 10 $\rangle$ +$\equiv$
  void $mp\_prompt\_file\_name(\mathbf{MP}\ mp, \mathbf{const\ char}\ *s, \mathbf{const\ char}\ *e)$;

873.   **void** *mp_prompt_file_name*(**MP** *mp*, **const char** ∗*s*, **const char** ∗*e*)
  {
    size_t *k*;   /∗ index into *buffer* ∗/
    char ∗*saved_cur_name*;
    if (*mp→interaction* ≡ *mp_scroll_mode*) *wake_up_terminal*( );
    if (*strcmp*(*s*, "input␣file␣name") ≡ 0) {
      mp_print_err(*mp*, "I␣can\'t␣open␣file␣'");
    }
    else {
      mp_print_err(*mp*, "I␣can\'t␣write␣on␣file␣'");
    }
    if (*strcmp*(*s*, "file␣name␣for␣output") ≡ 0) {
      mp_print(*mp*, *mp→name_of_file*);
    }
    else {
      mp_print_file_name(*mp*, *mp→cur_name*, *mp→cur_area*, *mp→cur_ext*);
    }
    mp_print(*mp*, "'.");
    if (*strcmp*(*e*, "") ≡ 0) *mp_show_context*(*mp*);
    mp_print_nl(*mp*, "Please␣type␣another␣");
    mp_print(*mp*, *s*);
    ;
    if (*mp→noninteractive* ∨ *mp→interaction* < *mp_scroll_mode*)
      mp_fatal_error(*mp*, "***␣(job␣aborted,␣file␣error␣in␣nonstop␣mode)");
    ;
    saved_cur_name = *xstrdup*(*mp→cur_name*);
    clear_terminal( );
    prompt_input(":␣");
    ⟨Scan file name in the buffer 874⟩;
    if (*strcmp*(*mp→cur_ext*, "") ≡ 0) *mp→cur_ext* = *xstrdup*(*e*);
    if (*strlen*(*mp→cur_name*) ≡ 0) {
      mp→cur_name = *saved_cur_name*;
    }
    else {
      xfree(*saved_cur_name*);
    }
    pack_cur_name;
  }

874.   ⟨Scan file name in the buffer 874⟩ ≡
  {
    mp_begin_name(*mp*);
    k = *mp→first*;
    while ((*mp→buffer*[*k*] ≡ '␣') ∧ (*k* < *mp→last*)) *incr*(*k*);
    while (1) {
      if (*k* ≡ *mp→last*) **break**;
      if (¬*mp_more_name*(*mp*, *mp→buffer*[*k*])) **break**;
      incr(*k*);
    }
    mp_end_name(*mp*);
  }
This code is used in section 873.

875.   The *open_log_file* routine is used to open the transcript file and to help it catch up to what has previously been printed on the terminal.

void *mp_open_log_file*(**MP** *mp*)
{
  unsigned *old_setting*;   /* previous *selector* setting */
  int *k*;   /* index into *months* and *buffer* */
  int *l*;   /* end of first input line */
  integer *m*;   /* the current month */
  const char *months = "JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC";
    /* abbreviations of month names */

  if (*mp*→*log_opened*) **return**;
  old_setting = *mp*→*selector*;
  if (*mp*→*job_name* ≡ Λ) {
    mp→*job_name* = *xstrdup*("mpout");
    ⟨Fix up *mp*→*internal*[*mp_job_name*] 868⟩;
  }
  mp_pack_job_name(*mp*, ".log");
  while (¬*mp_open_out*(*mp*, &*mp*→*log_file*, *mp_filetype_log*)) {
    ⟨Try to get a different log file name 877⟩;
  }
  mp→*log_name* = *xstrdup*(*mp*→*name_of_file*);
  mp→*selector* = *log_only*;
  mp→*log_opened* = *true*;
  ⟨Print the banner line, including the date and time 878⟩;
  mp→*input_stack*[*mp*→*input_ptr*] = *mp*→*cur_input*;   /* make sure bottom level is in memory */
  if (¬*mp*→*noninteractive*) {
    mp_print_nl(*mp*, "**");
    ;
    l = *mp*→*input_stack*[0].*limit_field* − 1;   /* last position of first line */
    for (*k* = 0; *k* ≤ *l*; *k*++) *mp_print_char*(*mp*, *mp*→*buffer*[*k*]);
    mp_print_ln(*mp*);   /* now the transcript file contains the first line of input */
  }
  mp→*selector* = *old_setting* + 2;   /* *log_only* or *term_and_log* */
}

876.   ⟨Dealloc variables 27⟩ +≡
  xfree(*mp*→*log_name*);

877.   Sometimes *open_log_file* is called at awkward moments when METAPOST is unable to print error messages or even to *show_context*. The *prompt_file_name* routine can result in a *fatal_error*, but the **error** routine will not be invoked because *log_opened* will be false.

The normal idea of *mp_batch_mode* is that nothing at all should be written on the terminal. However, in the unusual case that no log file could be opened, we make an exception and allow an explanatory message to be seen.

Incidentally, the program always refers to the log file as a '**transcript file**', because some systems cannot use the extension '.log' for this file.

⟨Try to get a different log file name 877⟩ ≡
```
{
  mp→selector = term_only;
  mp_prompt_file_name(mp, "transcript␣file␣name", ".log");
}
```
This code is used in section 875.

878.   ⟨Print the banner line, including the date and time 878⟩ ≡
```
{
  wlog(mp→banner);
  mp_print(mp, "␣␣");
  mp_print_int(mp, round_unscaled(internal_value(mp_day)));
  mp_print_char(mp, xord('␣'));
  m = round_unscaled(internal_value(mp_month));
  for (k = 3 * m − 3; k < 3 * m; k++) {
    wlog_chr((unsigned char) months[k]);
  }
  mp_print_char(mp, xord('␣'));
  mp_print_int(mp, round_unscaled(internal_value(mp_year)));
  mp_print_char(mp, xord('␣'));
  mp_print_dd(mp, round_unscaled(internal_value(mp_hour)));
  mp_print_char(mp, xord(':'));
  mp_print_dd(mp, round_unscaled(internal_value(mp_minute)));
}
```
This code is used in section 875.

879.   The *try_extension* function tries to open an input file determined by *cur_name*, *cur_area*, and the argument *ext*. It returns *false* if it can't find the file in *cur_area* or the appropriate system area.
```
static boolean mp_try_extension(MP mp, const char *ext)
{
  mp_pack_file_name(mp, mp→cur_name, mp→cur_area, ext);
  in_name = xstrdup(mp→cur_name);
  in_area = xstrdup(mp→cur_area);
  in_ext = xstrdup(ext);
  if (mp_open_in(mp, &cur_file, mp_filetype_program)) {
    return true;
  }
  else {
    mp_pack_file_name(mp, mp→cur_name, Λ, ext);
    return mp_open_in(mp, &cur_file, mp_filetype_program);
  }
}
```

880.   Let's turn now to the procedure that is used to initiate file reading when an '`input`' command is being processed.

```
void mp_start_input(MP mp)
{     /* METAPOST will input something */
  char *fname = Λ;
  ⟨ Put the desired file name in (cur_name, cur_ext, cur_area) 883 ⟩;
  while (1) {
    mp_begin_file_reading(mp);     /* set up cur_file and new level of input */
    if (strlen(mp→cur_ext) ≡ 0) {
      if (mp_try_extension(mp, ".mp")) break;
      else if (mp_try_extension(mp, "")) break;
      else if (mp_try_extension(mp, ".mf")) break;
    }
    else if (mp_try_extension(mp, mp→cur_ext)) {
      break;
    }
    mp_end_file_reading(mp);     /* remove the level that didn't work */
    mp_prompt_file_name(mp, "input␣file␣name", "");
  }
  name = mp_make_name_string(mp);
  fname = xstrdup(mp→name_of_file);
  if (mp→job_name ≡ Λ) {
    mp→job_name = xstrdup(mp→cur_name);
    ⟨ Fix up mp→internal[mp_job_name] 868 ⟩;
  }
  if (¬mp→log_opened) {
    mp_open_log_file(mp);
  }     /* open_log_file doesn't show_context, so limit and loc needn't be set to meaningful values yet */
  if (((int) mp→term_offset + (int) strlen(fname)) > (mp→max_print_line − 2)) mp_print_ln(mp);
  else if ((mp→term_offset > 0) ∨ (mp→file_offset > 0)) mp_print_char(mp, xord('␣'));
  mp_print_char(mp, xord('('));
  incr(mp→open_parens);
  mp_print(mp, fname);
  xfree(fname);
  update_terminal();
  ⟨ Flush name and replace it with cur_name if it won't be needed 881 ⟩;
  ⟨ Read the first line of the new file 882 ⟩;
}
```

881.   This code should be omitted if *make_name_string* returns something other than just a copy of its argument and the full file name is needed for opening `MPX` files or implementing the switch-to-editor option.

⟨ Flush *name* and replace it with *cur_name* if it won't be needed 881 ⟩ ≡
```
  mp_flush_string(mp, name);
  name = mp_rts(mp, mp→cur_name); xfree(mp→cur_name)
```
This code is used in section 880.

882.   If the file is empty, it is considered to contain a single blank line, so there is no need to test the
return value.

⟨ Read the first line of the new file 882 ⟩ ≡
   { **line** = 1;
   (**void**) *mp_input_ln*(*mp*, *cur_file*);
   mp_firm_up_the_line(*mp*);
   mp→*buffer*[*limit*] = *xord*(`'%'`);
   mp→*first* = (**size_t**)(*limit* + 1);
   loc = *start*; }

This code is used in sections 880 and 884.

883.   ⟨ Put the desired file name in (*cur_name*, *cur_ext*, *cur_area*) 883 ⟩ ≡
   while (*token_state* ∧ (*nloc* ≡ Λ)) *mp_end_token_list*(*mp*);
   if (*token_state*) {
      const char ∗*hlp*[] = {`"Sorry...I've␣converted␣what␣follows␣to␣tokens,"`,
          `"possibly␣garbaging␣the␣name␣you␣gave."`,
          `"Please␣delete␣the␣tokens␣and␣insert␣the␣name␣again."`, Λ};

      mp_error(*mp*, `"File␣names␣can't␣appear␣within␣macros"`, *hlp*, *true*);

      ;
   }
   if (*file_state*) {
      mp_scan_file_name(*mp*);
   }
   else {
      xfree(*mp*→*cur_name*);
      mp→*cur_name* = *xstrdup*(`""`);
      xfree(*mp*→*cur_ext*);
      mp→*cur_ext* = *xstrdup*(`""`);
      xfree(*mp*→*cur_area*);
      mp→*cur_area* = *xstrdup*(`""`);
   }

This code is used in section 880.

884.   The following simple routine starts reading the `MPX` file associated with the current input file.

> **void** $mp\_start\_mpx\_input(\textbf{MP}\ mp)$
> {
>   **char** $*origname = \Lambda$;   /* a copy of nameoffile */
>
> $mp\_pack\_file\_name(mp, in\_name, in\_area, in\_ext)$;
> $origname = xstrdup(mp\text{-}name\_of\_file)$;
> $mp\_pack\_file\_name(mp, in\_name, in\_area, \texttt{".mpx"})$;
>   **if** $(\neg(mp\text{-}run\_make\_mpx)(mp, origname, mp\text{-}name\_of\_file))$ **goto** `NOT_FOUND`;
> $mp\_begin\_file\_reading(mp)$;
>   **if** $(\neg mp\_open\_in(mp, \&cur\_file, mp\_filetype\_program))$ {
> $mp\_end\_file\_reading(mp)$;
>   **goto** `NOT_FOUND`;
>   }
> $name = mp\_make\_name\_string(mp)$;
> $mp\text{-}mpx\_name[iindex] = name$;
> $add\_str\_ref(name)$;
>   ⟨Read the first line of the new file 882⟩;
> $xfree(origname)$;
>   **return**;
> `NOT_FOUND`: ⟨Explain that the `MPX` file can't be read and $succumb$ 891⟩;
> $xfree(origname)$;
> }

885.   This should ideally be changed to do whatever is necessary to create the `MPX` file given by $name\_of\_file$ if it does not exist or if it is out of date. This requires invoking `MPtoTeX` on the $origname$ and passing the results through TeX and `DVItoMP`. (It is possible to use a completely different typesetting program if suitable postprocessor is available to perform the function of `DVItoMP`.)

886.   ⟨Exported types 15⟩ +≡
> **typedef int**$(*mp\_makempx\_cmd)(\textbf{MP}\ mp, \textbf{char}\ *origname, \textbf{char}\ *mtxname)$;

887.   ⟨Option variables 26⟩ +≡
> $mp\_makempx\_cmd\ run\_make\_mpx$;

888.   ⟨Allocate or initialize variables 28⟩ +≡
> $set\_callback\_option(run\_make\_mpx)$;

889.   ⟨Declarations 8⟩ +≡
> **static int** $mp\_run\_make\_mpx(\textbf{MP}\ mp, \textbf{char}\ *origname, \textbf{char}\ *mtxname)$;

890.   The default does nothing.

> **int** $mp\_run\_make\_mpx(\textbf{MP}\ mp, \textbf{char}\ *origname, \textbf{char}\ *mtxname)$
> {
>   (**void**) mp;
>   (**void**) $origname$;
>   (**void**) $mtxname$;
>   **return** $false$;
> }

891.   ⟨Explain that the MPX file can't be read and *succumb* 891⟩ ≡

  {

    const char ∗*hlp*[ ] = {"The␣two␣files␣given␣above␣are␣one␣of␣your␣source␣files",

        "and␣an␣auxiliary␣file␣I␣need␣to␣read␣to␣find␣out␣what␣your",

        "btex..etex␣blocks␣mean.␣If␣you␣don't␣know␣why␣I␣had␣trouble,",

        "try␣running␣it␣manually␣through␣MPtoTeX,␣TeX,␣and␣DVItoMP", Λ};

    if (*mp⃗interaction* ≡ *mp_error_stop_mode*) *wake_up_terminal*( );

    mp_print_nl(*mp*, ">>␣");

    mp_print(*mp*, *origname*);

    mp_print_nl(*mp*, ">>␣");

    mp_print(*mp*, *mp⃗name_of_file*);

    xfree(*origname*);

    if (*mp⃗interaction* ≡ *mp_error_stop_mode*) *mp⃗interaction* = *mp_scroll_mode*;

        /∗ no more interaction ∗/

    if (*mp⃗log_opened*) *mp_error*(*mp*, "!␣Unable␣to␣make␣mpx␣file", *hlp*, *true*);

    mp⃗history = *mp_fatal_error_stop*;

    mp_jump_out(*mp*);   /∗ irrecoverable error ∗/

  }

This code is used in section 884.

892.   The last file-opening commands are for files accessed via the **readfrom** operator and the **write** command. Such files are stored in separate arrays.

⟨Types in the outer block 33⟩ +≡

    typedef unsigned int readf_index;   /∗ 0..*max_read_files* ∗/

    typedef unsigned int write_index;   /∗ 0..*max_write_files* ∗/

893.   ⟨Global variables 14⟩ +≡

  readf_index *max_read_files*;   /∗ maximum number of simultaneously open **readfrom** files ∗/

  void ∗∗*rd_file*;   /∗ **readfrom** files ∗/

  char ∗∗*rd_fname*;   /∗ corresponding file name or 0 if file not open ∗/

  readf_index *read_files*;   /∗ number of valid entries in the above arrays ∗/

  write_index *max_write_files*;   /∗ maximum number of simultaneously open **write** ∗/

  void ∗∗*wr_file*;   /∗ **write** files ∗/

  char ∗∗*wr_fname*;   /∗ corresponding file name or 0 if file not open ∗/

  write_index *write_files*;   /∗ number of valid entries in the above arrays ∗/

894.   ⟨Allocate or initialize variables 28⟩ +≡

  mp⃗max_read_files = 8;

  mp⃗rd_file = *xmalloc*((*mp⃗max_read_files* + 1), **sizeof**(**void** ∗));

  mp⃗rd_fname = *xmalloc*((*mp⃗max_read_files* + 1), **sizeof**(**char** ∗));

  memset(*mp⃗rd_fname*, 0, **sizeof**(**char** ∗) ∗ (*mp⃗max_read_files* + 1));

  mp⃗max_write_files = 8;

  mp⃗wr_file = *xmalloc*((*mp⃗max_write_files* + 1), **sizeof**(**void** ∗));

  mp⃗wr_fname = *xmalloc*((*mp⃗max_write_files* + 1), **sizeof**(**char** ∗));

  memset(*mp⃗wr_fname*, 0, **sizeof**(**char** ∗) ∗ (*mp⃗max_write_files* + 1));

895.   This routine starts reading the file named by string s without setting *loc*, *limit*, or *name*. It returns *false* if the file is empty or cannot be opened. Otherwise it updates *rd_file*[n] and *rd_fname*[n].

> **static boolean** *mp_start_read_input*(**MP** *mp*, **char** *∗s*, **readf_index** *n*)
> {
>   *mp_ptr_scan_file*(*mp*, *s*);
>   *pack_cur_name*;
>   *mp_begin_file_reading*(*mp*);
>   **if** (¬*mp_open_in*(*mp*, &*mp*→*rd_file*[n], (**int**)(*mp_filetype_text* + n))) **goto** NOT_FOUND;
>   **if** (¬*mp_input_ln*(*mp*, *mp*→*rd_file*[n])) {
>   (*mp*→*close_file*)(*mp*, *mp*→*rd_file*[n]);
>   **goto** NOT_FOUND;
>   }
>   *mp*→*rd_fname*[n] = *xstrdup*(*s*);
>   **return** *true*;
> NOT_FOUND: *mp_end_file_reading*(*mp*);
>   **return** *false*;
> }

896.   Open *wr_file*[n] using file name s and update *wr_fname*[n].

⟨ Declarations 8 ⟩ +≡
> **static void** *mp_open_write_file*(**MP** *mp*, **char** *∗s*, **readf_index** *n*);

897.   **void** *mp_open_write_file*(**MP** *mp*, **char** *∗s*, **readf_index** *n*)
> {
>   *mp_ptr_scan_file*(*mp*, *s*);
>   *pack_cur_name*;
>   **while** (¬*mp_open_out*(*mp*, &*mp*→*wr_file*[n], (**int**)(*mp_filetype_text* + n)))
>   *mp_prompt_file_name*(*mp*, "file␣name␣for␣write␣output", "");
>   *mp*→*wr_fname*[n] = *xstrdup*(*s*);
> }

898.   Introduction to the parsing routines.   We come now to the central nervous system that sparks many of METAPOST's activities. By evaluating expressions, from their primary constituents to ever larger subexpressions, METAPOST builds the structures that ultimately define complete pictures or fonts of type.

Four mutually recursive subroutines are involved in this process: We call them

$$scan\_primary, \ scan\_secondary, \ scan\_tertiary, \ \text{and} \ scan\_expression.$$

Each of them is parameterless and begins with the first token to be scanned already represented in cur_cmd, cur_mod, and cur_sym. After execution, the value of the primary or secondary or tertiary or expression that was found will appear in the global variables cur_type and cur_exp. The token following the expression will be represented in cur_cmd, cur_mod, and cur_sym.

Technically speaking, the parsing algorithms are "LL(1)," more or less; backup mechanisms have been added in order to provide reasonable error recovery.

#define $cur\_exp\_value\_boolean()$ $number\_to\_int(mp\text{-}cur\_exp.data.n)$
#define $cur\_exp\_value\_number()$ $mp\text{-}cur\_exp.data.n$
#define $cur\_exp\_node()$ $mp\text{-}cur\_exp.data.node$
#define $cur\_exp\_str()$ $mp\text{-}cur\_exp.data.str$
#define $cur\_exp\_knot()$ $mp\text{-}cur\_exp.data.p$
#define $set\_cur\_exp\_value\_scaled(A)$ **do**
　　　{
　　　　if $(cur\_exp\_str())$ {
　　　　　$delete\_str\_ref(cur\_exp\_str());$
　　　　}
　　　　$set\_number\_from\_scaled(mp\text{-}cur\_exp.data.n,(A));$
　　　　$cur\_exp\_node() = \Lambda;$
　　　　$cur\_exp\_str() = \Lambda;$
　　　　$cur\_exp\_knot() = \Lambda;$
　　　}
　　　while (0)
#define $set\_cur\_exp\_value\_boolean(A)$ **do**
　　　{
　　　　if $(cur\_exp\_str())$ {
　　　　　$delete\_str\_ref(cur\_exp\_str());$
　　　　}
　　　　$set\_number\_from\_int(mp\text{-}cur\_exp.data.n,(A));$
　　　　$cur\_exp\_node() = \Lambda;$
　　　　$cur\_exp\_str() = \Lambda;$
　　　　$cur\_exp\_knot() = \Lambda;$
　　　}
　　　while (0)
#define $set\_cur\_exp\_value\_number(A)$ **do**
　　　{
　　　　if $(cur\_exp\_str())$ {
　　　　　$delete\_str\_ref(cur\_exp\_str());$
　　　　}
　　　　$number\_clone(mp\text{-}cur\_exp.data.n,(A));$
　　　　$cur\_exp\_node() = \Lambda;$
　　　　$cur\_exp\_str() = \Lambda;$
　　　　$cur\_exp\_knot() = \Lambda;$
　　　}
　　　while (0)
#define $set\_cur\_exp\_node(A)$ **do**

```
          {
            if (cur_exp_str( )) {
                delete_str_ref (cur_exp_str ( ));
            }
            cur_exp_node( ) = A;
            cur_exp_str( ) = Λ;
            cur_exp_knot( ) = Λ;
            set_number_to_zero(mp→cur_exp.data.n);
          }
          while (0)
#define  set_cur_exp_str (A)  do
          {
            if (cur_exp_str( )) {
                delete_str_ref (cur_exp_str ( ));
            }
            cur_exp_str( ) = A;
            add_str_ref (cur_exp_str ( ));
            cur_exp_node( ) = Λ;
            cur_exp_knot( ) = Λ;
            set_number_to_zero(mp→cur_exp.data.n);
          }
          while (0)
#define  set_cur_exp_knot (A)  do
          {
            if (cur_exp_str( )) {
                delete_str_ref (cur_exp_str ( ));
            }
            cur_exp_knot( ) = A;
            cur_exp_node( ) = Λ;
            cur_exp_str( ) = Λ;
            set_number_to_zero(mp→cur_exp.data.n);
          }
          while (0)
```

899.   ⟨Global variables 14⟩ +≡
  mp_value *cur_exp*;   /∗ the value of the expression just found ∗/

900.   ⟨Set initial values of key variables 38⟩ +≡
  memset(&*mp→cur_exp.data*, 0, **sizeof**(**mp_value**));
  new_number(*mp→cur_exp.data.n*);

901.   ⟨Free table entries 183⟩ +≡
  free_number(*mp→cur_exp.data.n*);

902.   Many different kinds of expressions are possible, so it is wise to have precise descriptions of what *cur_type* and *cur_exp* mean in all cases:

cur_type = *mp_vacuous* means that this expression didn't turn out to have a value at all, because it arose from a **begingroup** ... **endgroup** construction in which there was no expression before the **endgroup**. In this case *cur_exp* has some irrelevant value.

cur_type = *mp_boolean_type* means that *cur_exp* is either *true_code* or *false_code*.

cur_type = *mp_unknown_boolean* means that *cur_exp* points to a capsule node that is in a ring of equivalent booleans whose value has not yet been defined.

cur_type = *mp_string_type* means that *cur_exp* is a string number (i.e., an integer in the range $0 \le cur\_exp < str\_ptr$). That string's reference count includes this particular reference.

cur_type = *mp_unknown_string* means that *cur_exp* points to a capsule node that is in a ring of equivalent strings whose value has not yet been defined.

cur_type = *mp_pen_type* means that *cur_exp* points to a node in a pen. Nobody else points to any of the nodes in this pen. The pen may be polygonal or elliptical.

cur_type = *mp_unknown_pen* means that *cur_exp* points to a capsule node that is in a ring of equivalent pens whose value has not yet been defined.

cur_type = *mp_path_type* means that *cur_exp* points to a the first node of a path; nobody else points to this particular path. The control points of the path will have been chosen.

cur_type = *mp_unknown_path* means that *cur_exp* points to a capsule node that is in a ring of equivalent paths whose value has not yet been defined.

cur_type = *mp_picture_type* means that *cur_exp* points to an edge header node. There may be other pointers to this particular set of edges. The header node contains a reference count that includes this particular reference.

cur_type = *mp_unknown_picture* means that *cur_exp* points to a capsule node that is in a ring of equivalent pictures whose value has not yet been defined.

cur_type = *mp_transform_type* means that *cur_exp* points to a *mp_transform_type* capsule node. The *value* part of this capsule points to a transform node that contains six numeric values, each of which is *independent*, *dependent*, *mp_proto_dependent*, or *known*.

cur_type = *mp_color_type* means that *cur_exp* points to a *color_type* capsule node. The *value* part of this capsule points to a color node that contains three numeric values, each of which is *independent*, *dependent*, *mp_proto_dependent*, or *known*.

cur_type = *mp_cmykcolor_type* means that *cur_exp* points to a *mp_cmykcolor_type* capsule node. The *value* part of this capsule points to a color node that contains four numeric values, each of which is *independent*, *dependent*, *mp_proto_dependent*, or *known*.

cur_type = *mp_pair_type* means that *cur_exp* points to a capsule node whose type is *mp_pair_type*. The *value* part of this capsule points to a pair node that contains two numeric values, each of which is *independent*, *dependent*, *mp_proto_dependent*, or *known*.

cur_type = *mp_known* means that *cur_exp* is a *scaled* value.

cur_type = *mp_dependent* means that *cur_exp* points to a capsule node whose type is *dependent*. The *dep_list* field in this capsule points to the associated dependency list.

cur_type = *mp_proto_dependent* means that *cur_exp* points to a *mp_proto_dependent* capsule node. The *dep_list* field in this capsule points to the associated dependency list.

cur_type = *independent* means that *cur_exp* points to a capsule node whose type is *independent*. This somewhat unusual case can arise, for example, in the expression 'x + **begingroup string** x; 0 **endgroup**'.

cur_type = *mp_token_list* means that *cur_exp* points to a linked list of tokens.

The possible settings of *cur_type* have been listed here in increasing numerical order. Notice that *cur_type* will never be *mp_numeric_type* or *suffixed_macro* or *mp_unsuffixed_macro*, although variables of those types are allowed. Conversely, METAPOST has no variables of type *mp_vacuous* or *token_list*.

903.   Capsules are non-symbolic nodes that have a similar meaning to *cur_type* and *cur_exp*. Such nodes have *name_type* = *capsule*, and their *type* field is one of the possibilities for *cur_type* listed above. Also *link* ≤ **void** in capsules that aren't part of a token list.

The *value* field of a capsule is, in most cases, the value that corresponds to its *type*, as *cur_exp* corresponds to *cur_type*. However, when *cur_exp* would point to a capsule, no extra layer of indirection is present; the *value* field is what would have been called *value*(*cur_exp*) if it had not been encapsulated. Furthermore, if the type is *dependent* or *mp_proto_dependent*, the *value* field of a capsule is replaced by *dep_list* and *prev_dep* fields, since dependency lists in capsules are always part of the general *dep_list* structure.

The *get_x_next* routine is careful not to change the values of *cur_type* and *cur_exp* when it gets an expanded token. However, *get_x_next* might call a macro, which might parse an expression, which might execute lots of commands in a group; hence it's possible that *cur_type* might change from, say, *mp_unknown_boolean* to *mp_boolean_type*, or from *dependent* to *known* or *independent*, during the time *get_x_next* is called. The programs below are careful to stash sensitive intermediate results in capsules, so that METAPOST's generality doesn't cause trouble.

Here's a procedure that illustrates these conventions. It takes the contents of (*cur_type*, *cur_exp*) and stashes them away in a capsule. It is not used when *cur_type* = *mp_token_list*. After the operation, *cur_type* = *mp_vacuous*; hence there is no need to copy path lists or to update reference counts, etc.

The special link `MP_VOID` is put on the capsule returned by *stash_cur_exp*, because this procedure is used to store macro parameters that must be easily distinguishable from token lists.

⟨ Declare the stashing/unstashing routines 903 ⟩ ≡
  static mp_node *mp_stash_cur_exp*(**MP** *mp*)
  {
    mp_node *p*;   /∗ the capsule that will be returned ∗/
    mp_variable_type exp_type = *mp*→*cur_exp.type*;
    switch (*exp_type*) {
    case *unknown_types*: **case** *mp_transform_type*: **case** *mp_color_type*: **case** *mp_pair_type*:
      case *mp_dependent*: **case** *mp_proto_dependent*: **case** *mp_independent*: **case** *mp_cmykcolor_type*:
      p = *cur_exp_node*( );
      break;   /∗ **case** *mp_path_type*: **case** *mp_pen_type*: **case** *mp_string_type*: ∗/
    default: *p* = *mp_get_value_node*(*mp*);
      mp_name_type(*p*) = *mp_capsule*;
      mp_type(*p*) = *mp*→*cur_exp.type*;
      set_value_number(*p*, *cur_exp_value_number*( ));   /∗ this also resets the rest to 0/NULL ∗/
      if (*cur_exp_str*( )) {
        set_value_str(*p*, *cur_exp_str*( ));
      }
      else if (*cur_exp_knot*( )) {
        set_value_knot(*p*, *cur_exp_knot*( ));
      }
      else if (*cur_exp_node*( )) {
        set_value_node(*p*, *cur_exp_node*( ));
      }
      break;
    }
    mp→*cur_exp.type* = *mp_vacuous*;
    mp_link(*p*) = `MP_VOID`;
    return *p*;
  }

See also section 904.

This code is used in section 906.

904.   The inverse of *stash_cur_exp* is the following procedure, which deletes an unnecessary capsule and puts its contents into *cur_type* and *cur_exp*.

The program steps of METAPOST can be divided into two categories: those in which *cur_type* and *cur_exp* are "alive" and those in which they are "dead," in the sense that *cur_type* and *cur_exp* contain relevant information or not. It's important not to ignore them when they're alive, and it's important not to pay attention to them when they're dead.

There's also an intermediate category: If *cur_type* = *mp_vacuous*, then *cur_exp* is irrelevant, hence we can proceed without caring if *cur_type* and *cur_exp* are alive or dead. In such cases we say that *cur_type* and *cur_exp* are *dormant*. It is permissible to call *get_x_next* only when they are alive or dormant.

The *stash* procedure above assumes that *cur_type* and *cur_exp* are alive or dormant. The *unstash* procedure assumes that they are dead or dormant; it resuscitates them.

⟨ Declare the stashing/unstashing routines 903 ⟩ +≡
   static void *mp_unstash_cur_exp*(**MP** *mp*, **mp_node** *p*);

905.   **void** *mp_unstash_cur_exp*(**MP** *mp*, **mp_node** *p*)
  {
    mp→cur_exp.type = *mp_type*(*p*);
    switch (*mp→cur_exp.type*) {
    case *unknown_types*: **case** *mp_transform_type*: **case** *mp_color_type*: **case** *mp_pair_type*:
      case *mp_dependent*: **case** *mp_proto_dependent*: **case** *mp_independent*: **case** *mp_cmykcolor_type*:
      set_cur_exp_node(*p*);
      break;
    case *mp_token_list*:   /∗ this is how symbols are stashed ∗/
      set_cur_exp_node(*value_node*(*p*));
      mp_free_value_node(*mp*, *p*);
      break;
    case *mp_path_type*: **case** *mp_pen_type*: *set_cur_exp_knot*(*value_knot*(*p*));
      mp_free_value_node(*mp*, *p*);
      break;
    case *mp_string_type*: *set_cur_exp_str*(*value_str*(*p*));
      mp_free_value_node(*mp*, *p*);
      break;
    case *mp_picture_type*: *set_cur_exp_node*(*value_node*(*p*));
      mp_free_value_node(*mp*, *p*);
      break;
    case *mp_boolean_type*: **case** *mp_known*: *set_cur_exp_value_number*(*value_number*(*p*));
      mp_free_value_node(*mp*, *p*);
      break;
    default: *set_cur_exp_value_number*(*value_number*(*p*));
      if (*value_knot*(*p*)) {
        set_cur_exp_knot(*value_knot*(*p*));
      }
      else if (*value_node*(*p*)) {
        set_cur_exp_node(*value_node*(*p*));
      }
      else if (*value_str*(*p*)) {
        set_cur_exp_str(*value_str*(*p*));
      }
      mp_free_value_node(*mp*, *p*);
      break;
    }
  }

906.   The following procedure prints the values of expressions in an abbreviated format. If its first parameter p is NULL, the value of (cur_type, cur_exp) is displayed; otherwise p should be a capsule containing the desired value. The second parameter controls the amount of output. If it is 0, dependency lists will be abbreviated to '`linearform`' unless they consist of a single term. If it is greater than 1, complicated structures (pens, pictures, and paths) will be displayed in full.

⟨ Declarations 8 ⟩ +≡
    ⟨ Declare the procedure called $print\_dp$ 915 ⟩;
    ⟨ Declare the stashing/unstashing routines 903 ⟩;

  static void mp_print_exp(**MP** mp, **mp_node** p, **quarterword** $verbosity$);

907.   **void** mp_print_exp(**MP** mp, **mp_node** p, **quarterword** $verbosity$)
  {
    boolean $restore\_cur\_exp$;   /∗ should cur_exp be restored? ∗/
    $mp\_variable\_type\,t$;   /∗ the type of the expression ∗/
    mp_number vv;   /∗ the value of the expression ∗/
    mp_node $v = \Lambda$;
    $new\_number(vv)$;
    if $(p \neq \Lambda)$ {
      $restore\_cur\_exp = false$;
    }
    else {
      $p = mp\_stash\_cur\_exp(mp)$;
      $restore\_cur\_exp = true$;
    }
    $t = mp\_type(p)$;
    if $(t < mp\_dependent)$ {   /∗ no dep list, could be a capsule ∗/
      if $(t \neq mp\_vacuous \wedge t \neq mp\_known \wedge value\_node(p) \neq \Lambda)$ $v = value\_node(p)$;
      else $number\_clone(vv, value\_number(p))$;
    }
    else if $(t < mp\_independent)$ {
      $v = $ (**mp_node**) dep_list((**mp_value_node**) p);
    }
    ⟨ Print an abbreviated value of v or vv with format depending on t 908 ⟩;
    if $(restore\_cur\_exp)$ $mp\_unstash\_cur\_exp(mp, p)$;
    $free\_number(vv)$;
  }

908.   ⟨Print an abbreviated value of v or vv with format depending on t 908⟩ ≡
  switch (t) {
  case *mp_vacuous*: *mp_print*(*mp*, "vacuous");
    break;
  case *mp_boolean_type*:
    if (*number_to_boolean*(*vv*) ≡ *mp_true_code*) *mp_print*(*mp*, "true");
    else *mp_print*(*mp*, "false");
    break;
  case *unknown_types*: **case** *mp_numeric_type*:
    ⟨Display a variable that's been declared but not defined 916⟩;
    break;
  case *mp_string_type*: *mp_print_char*(*mp*, *xord*('"'));
  mp_print_str(*mp*, *value_str*(*p*));
  mp_print_char(*mp*, *xord*('"'));
    break;
  case *mp_pen_type*: **case** *mp_path_type*: **case** *mp_picture_type*: ⟨Display a complex type 914⟩;
    break;
  case *mp_transform_type*:
    if (*number_zero*(*vv*) ∧ *v* ≡ Λ) *mp_print_type*(*mp*, *t*);
    else ⟨Display a transform node 911⟩;
    break;
  case *mp_color_type*:
    if (*number_zero*(*vv*) ∧ *v* ≡ Λ) *mp_print_type*(*mp*, *t*);
    else ⟨Display a color node 912⟩;
    break;
  case *mp_pair_type*:
    if (*number_zero*(*vv*) ∧ *v* ≡ Λ) *mp_print_type*(*mp*, *t*);
    else ⟨Display a pair node 910⟩;
    break;
  case *mp_cmykcolor_type*:
    if (*number_zero*(*vv*) ∧ *v* ≡ Λ) *mp_print_type*(*mp*, *t*);
    else ⟨Display a cmykcolor node 913⟩;
    break;
  case *mp_known*: *print_number*(*vv*);
    break;
  case *mp_dependent*: **case** *mp_proto_dependent*: *mp_print_dp*(*mp*, *t*, (**mp_value_node**) *v*, *verbosity*);
    break;
  case *mp_independent*: *mp_print_variable_name*(*mp*, *p*);
    break;
  default: *mp_confusion*(*mp*, "exp");
    break;
  }
This code is used in section 907.

909.   ⟨Display big node item v 909⟩ ≡
  {
    if (*mp_type*(*v*) ≡ *mp_known*) *print_number*(*value_number*(*v*));
    else if (*mp_type*(*v*) ≡ *mp_independent*) *mp_print_variable_name*(*mp*, *v*);
    else *mp_print_dp*(*mp*, *mp_type*(*v*), (**mp_value_node**) *dep_list*((**mp_value_node**) *v*), *verbosity*);
  }
This code is used in sections 910, 911, 912, and 913.

910.   In these cases, v starts as the big node.

⟨ Display a pair node 910 ⟩ ≡
  {
    mp_node $vvv = v$;
    $mp\_print\_char(mp, xord(\texttt{'('}))$;   /∗ clang: dereference of null pointer ∗/
    $assert(vvv)$;
    $v = x\_part(vvv)$;
    ⟨ Display big node item v 909 ⟩;
    $mp\_print\_char(mp, xord(\texttt{','}))$;
    $v = y\_part(vvv)$;
    ⟨ Display big node item v 909 ⟩;
    $mp\_print\_char(mp, xord(\texttt{')'}))$;
  }

This code is used in section 908.

911.   ⟨ Display a transform node 911 ⟩ ≡
  {
    mp_node $vvv = v$;
    $mp\_print\_char(mp, xord(\texttt{'('}))$;   /∗ clang: dereference of null pointer ∗/
    $assert(vvv)$;
    $v = tx\_part(vvv)$;
    ⟨ Display big node item v 909 ⟩;
    $mp\_print\_char(mp, xord(\texttt{','}))$;
    $v = ty\_part(vvv)$;
    ⟨ Display big node item v 909 ⟩;
    $mp\_print\_char(mp, xord(\texttt{','}))$;
    $v = xx\_part(vvv)$;
    ⟨ Display big node item v 909 ⟩;
    $mp\_print\_char(mp, xord(\texttt{','}))$;
    $v = xy\_part(vvv)$;
    ⟨ Display big node item v 909 ⟩;
    $mp\_print\_char(mp, xord(\texttt{','}))$;
    $v = yx\_part(vvv)$;
    ⟨ Display big node item v 909 ⟩;
    $mp\_print\_char(mp, xord(\texttt{','}))$;
    $v = yy\_part(vvv)$;
    ⟨ Display big node item v 909 ⟩;
    $mp\_print\_char(mp, xord(\texttt{')'}))$;
  }

This code is used in section 908.

912.   ⟨Display a color node 912⟩ ≡
  {
     mp_node $vvv = v$;
     $mp\_print\_char(mp, xord('('));$ /∗ clang: dereference of null pointer ∗/
     $assert(vvv)$;
     $v = red\_part(vvv)$;
     ⟨Display big node item v 909⟩;
     $mp\_print\_char(mp, xord(','))$;
     $v = green\_part(vvv)$;
     ⟨Display big node item v 909⟩;
     $mp\_print\_char(mp, xord(','))$;
     $v = blue\_part(vvv)$;
     ⟨Display big node item v 909⟩;
     $mp\_print\_char(mp, xord(')'))$;
  }
This code is used in section 908.

913.   ⟨Display a cmykcolor node 913⟩ ≡
  {
     mp_node $vvv = v$;
     $mp\_print\_char(mp, xord('('));$ /∗ clang: dereference of null pointer ∗/
     $assert(vvv)$;
     $v = cyan\_part(vvv)$;
     ⟨Display big node item v 909⟩;
     $mp\_print\_char(mp, xord(','))$;
     $v = magenta\_part(vvv)$;
     ⟨Display big node item v 909⟩;
     $mp\_print\_char(mp, xord(','))$;
     $v = yellow\_part(vvv)$;
     ⟨Display big node item v 909⟩;
     $mp\_print\_char(mp, xord(','))$;
     $v = black\_part(vvv)$;
     ⟨Display big node item v 909⟩;
     $mp\_print\_char(mp, xord(')'))$;
  }
This code is used in section 908.

914.   Values of type **picture**, **path**, and **pen** are displayed verbosely in the log file only, unless the user
has given a positive value to *tracingonline*.

⟨ Display a complex type 914 ⟩ ≡

 if (*verbosity* ≤ 1) {

  mp_print_type(*mp*, *t*);

 }

 else {

  if (*mp*→*selector* ≡ *term_and_log*)

   if (*number_nonpositive*(*internal_value*(*mp_tracing_online*))) {

    mp→*selector* = *term_only*;

    mp_print_type(*mp*, *t*);

    mp_print(*mp*, "␣(see␣the␣transcript␣file)");

    mp→*selector* = *term_and_log*;

   }

  ;

  switch (*t*) {

  case *mp_pen_type*: *mp_print_pen*(*mp*, *value_knot*(*p*), "", *false*);

   break;

  case *mp_path_type*: *mp_print_path*(*mp*, *value_knot*(*p*), "", *false*);

   break;

  case *mp_picture_type*: *mp_print_edges*(*mp*, *v*, "", *false*);

   break;

  default: **break**;

  }

 }

This code is used in section 908.

915.   ⟨ Declare the procedure called *print_dp* 915 ⟩ ≡

 static void *mp_print_dp*(**MP** *mp*, **quarterword** *t*, **mp_value_node** *p*, **quarterword** *verbosity*)

 {

  mp_value_node *q*; /∗ the node following *p* ∗/

  q = (**mp_value_node**) *mp_link*(*p*);

  if ((*dep_info*(*q*) ≡ Λ) ∨ (*verbosity* > 0)) *mp_print_dependency*(*mp*, *p*, *t*);

  else *mp_print*(*mp*, "linearform");

 }

This code is used in section 906.

916.   The displayed name of a variable in a ring will not be a capsule unless the ring consists entirely of
capsules.

⟨ Display a variable that's been declared but not defined 916 ⟩ ≡

 {

  mp_print_type(*mp*, *t*);

  if (*v* ≠ Λ) {

   mp_print_char(*mp*, *xord*('␣'));

   while ((*mp_name_type*(*v*) ≡ *mp_capsule*) ∧ (*v* ≠ *p*)) *v* = *value_node*(*v*);

   mp_print_variable_name(*mp*, *v*);

  }

  ;

 }

This code is used in section 908.

917.   When errors are detected during parsing, it is often helpful to display an expression just above the
error message, using *disp_err* just before *mp_error*.

⟨ Declarations 8 ⟩ +≡
  static void *mp_disp_err*(**MP** *mp*, **mp_node** *p*);

918.   **void** *mp_disp_err*(**MP** *mp*, **mp_node** *p*)
  {
    if (*mp*→*interaction* ≡ *mp_error_stop_mode*) *wake_up_terminal*( );
    mp_print_nl(*mp*, ">>␣");
    ;
    mp_print_exp(*mp*, *p*, 1);   /∗ "medium verbose" printing of the expression ∗/
  }

919.   If *cur_type* and *cur_exp* contain relevant information that should be recycled, we will use the following
procedure, which changes *cur_type* to *known* and stores a given value in *cur_exp*. We can think of *cur_type*
and *cur_exp* as either alive or dormant after this has been done, because *cur_exp* will not contain a pointer
value.

920.   **void** *mp_flush_cur_exp*(**MP** *mp*, **mp_value** *v*)
  {
    if (*is_number*(*mp*→*cur_exp*.*data*.*n*)) {
      free_number(*mp*→*cur_exp*.*data*.*n*);
    }
    switch (*mp*→*cur_exp*.*type*) {
    case *unknown_types*: **case** *mp_transform_type*: **case** *mp_color_type*: **case** *mp_pair_type*:
      case *mp_dependent*: **case** *mp_proto_dependent*: **case** *mp_independent*: **case** *mp_cmykcolor_type*:
      mp_recycle_value(*mp*, *cur_exp_node*( ));
      mp_free_value_node(*mp*, *cur_exp_node*( ));
      break;
    case *mp_string_type*: *delete_str_ref*(*cur_exp_str*( ));
      break;
    case *mp_pen_type*: **case** *mp_path_type*: *mp_toss_knot_list*(*mp*, *cur_exp_knot*( ));
      break;
    case *mp_picture_type*: *delete_edge_ref*(*cur_exp_node*( ));
      break;
    default: **break**;
    }
    mp→*cur_exp* = *v*;
    mp→*cur_exp*.*type* = *mp_known*;
  }

921.   There's a much more general procedure that is capable of releasing the storage associated with any
non-symbolic value packet.

⟨ Declarations 8 ⟩ +≡
  static void *mp_recycle_value*(**MP** *mp*, **mp_node** *p*);

922.   **static void** *mp_recycle_value* (**MP** *mp*, **mp_node** *p*)
{
  mp_variable_type t;   /* a type code */
  FUNCTION_TRACE2("mp_recycle_value(%p)\n", *p*);
  t = *mp_type* (*p*);
  switch (*t*) {
  case *mp_vacuous*: **case** *mp_boolean_type*: **case** *mp_known*: **case** *mp_numeric_type*: **break**;
  case *unknown_types*: *mp_ring_delete* (*mp*, *p*);
    break;
  case *mp_string_type*: *delete_str_ref* (*value_str* (*p*));
    break;
  case *mp_path_type*: **case** *mp_pen_type*: *mp_toss_knot_list* (*mp*, *value_knot* (*p*));
    break;
  case *mp_picture_type*: *delete_edge_ref* (*value_node* (*p*));
    break;
  case *mp_cmykcolor_type*:
    if (*value_node* (*p*) ≠ Λ) {
      mp_recycle_value (*mp*, *cyan_part* (*value_node* (*p*)));
      mp_recycle_value (*mp*, *magenta_part* (*value_node* (*p*)));
      mp_recycle_value (*mp*, *yellow_part* (*value_node* (*p*)));
      mp_recycle_value (*mp*, *black_part* (*value_node* (*p*)));
      mp_free_value_node (*mp*, *cyan_part* (*value_node* (*p*)));
      mp_free_value_node (*mp*, *magenta_part* (*value_node* (*p*)));
      mp_free_value_node (*mp*, *black_part* (*value_node* (*p*)));
      mp_free_value_node (*mp*, *yellow_part* (*value_node* (*p*)));
      mp_free_node (*mp*, *value_node* (*p*), *cmykcolor_node_size*);
    }
    break;
  case *mp_pair_type*:
    if (*value_node* (*p*) ≠ Λ) {
      mp_recycle_value (*mp*, *x_part* (*value_node* (*p*)));
      mp_recycle_value (*mp*, *y_part* (*value_node* (*p*)));
      mp_free_value_node (*mp*, *x_part* (*value_node* (*p*)));
      mp_free_value_node (*mp*, *y_part* (*value_node* (*p*)));
      mp_free_pair_node (*mp*, *value_node* (*p*));
    }
    break;
  case *mp_color_type*:
    if (*value_node* (*p*) ≠ Λ) {
      mp_recycle_value (*mp*, *red_part* (*value_node* (*p*)));
      mp_recycle_value (*mp*, *green_part* (*value_node* (*p*)));
      mp_recycle_value (*mp*, *blue_part* (*value_node* (*p*)));
      mp_free_value_node (*mp*, *red_part* (*value_node* (*p*)));
      mp_free_value_node (*mp*, *green_part* (*value_node* (*p*)));
      mp_free_value_node (*mp*, *blue_part* (*value_node* (*p*)));
      mp_free_node (*mp*, *value_node* (*p*), *color_node_size*);
    }
    break;
  case *mp_transform_type*:
    if (*value_node* (*p*) ≠ Λ) {
      mp_recycle_value (*mp*, *tx_part* (*value_node* (*p*)));
      mp_recycle_value (*mp*, *ty_part* (*value_node* (*p*)));

    $mp\_recycle\_value\,(mp, xx\_part\,(value\_node\,(p)));$
    $mp\_recycle\_value\,(mp, xy\_part\,(value\_node\,(p)));$
    $mp\_recycle\_value\,(mp, yx\_part\,(value\_node\,(p)));$
    $mp\_recycle\_value\,(mp, yy\_part\,(value\_node\,(p)));$
    $mp\_free\_value\_node\,(mp, tx\_part\,(value\_node\,(p)));$
    $mp\_free\_value\_node\,(mp, ty\_part\,(value\_node\,(p)));$
    $mp\_free\_value\_node\,(mp, xx\_part\,(value\_node\,(p)));$
    $mp\_free\_value\_node\,(mp, xy\_part\,(value\_node\,(p)));$
    $mp\_free\_value\_node\,(mp, yx\_part\,(value\_node\,(p)));$
    $mp\_free\_value\_node\,(mp, yy\_part\,(value\_node\,(p)));$
    $mp\_free\_node\,(mp, value\_node\,(p), transform\_node\_size);$
  }
  break;
case $mp\_dependent$: **case** $mp\_proto\_dependent$:   /∗ Recycle a dependency list ∗/
  {
    mp_value_node $qq = ($**mp_value_node**$)\ dep\_list\,(($**mp_value_node**$)\ p);$

    while $(dep\_info\,(qq) \neq \Lambda)\ \ qq = ($**mp_value_node**$)\ mp\_link\,(qq);$
    $set\_mp\_link\,(prev\_dep\,(($**mp_value_node**$)\ p), mp\_link\,(qq));$
    $set\_prev\_dep\,(mp\_link\,(qq), prev\_dep\,(($**mp_value_node**$)\ p));$
    $set\_mp\_link\,(qq, \Lambda);$
    $mp\_flush\_node\_list\,(mp, ($**mp_node**$)\ dep\_list\,(($**mp_value_node**$)\ p));$
  }
  break;
case $mp\_independent$: ⟨ Recycle an independent variable 923 ⟩;
  break;
case mp_token_list: **case** $mp\_structured$: $mp\_confusion\,(mp,$ "recycle"$);$
  break;
case $mp\_unsuffixed\_macro$: **case** $mp\_suffixed\_macro$: $mp\_delete\_mac\_ref\,(mp, value\_node\,(p));$
  break;
default: /∗ there are no other valid cases, but please the compiler ∗/
  break;
  }
  $mp\_type\,(p) = mp\_undefined;$
}

923.   When an independent variable disappears, it simply fades away, unless something depends on it. In the latter case, a dependent variable whose coefficient of dependence is maximal will take its place. The relevant algorithm is due to Ignacio A. Zabala, who implemented it as part of his Ph.n-¿data. thesis (Stanford University, December 1982).

For example, suppose that variable x is being recycled, and that the only variables depending on x are $y = 2x + a$ and $z = x + b$. In this case we want to make y independent and $z = .5y - .5a + b$; no other variables will depend on y. If *tracingequations* > 0 in this situation, we will print '`### -2x=-y+a`'.

There's a slight complication, however: An independent variable x can occur both in dependency lists and in proto-dependency lists. This makes it necessary to be careful when deciding which coefficient is maximal.

Furthermore, this complication is not so slight when a proto-dependent variable is chosen to become independent. For example, suppose that $y = 2x + 100a$ is proto-dependent while $z = x + b$ is dependent; then we must change $z = .5y - 50a + b$ to a proto-dependency, because of the large coefficient '50'.

In order to deal with these complications without wasting too much time, we shall link together the occurrences of x among all the linear dependencies, maintaining separate lists for the dependent and proto-dependent cases.

⟨ Recycle an independent variable 923 ⟩ ≡
```
{
  mp_value_node q, r, s;
  mp_node pp;       /* link manipulation register */
  mp_number v;       /* a value */
  mp_number test;      /* a temporary value */

  new_number(test);
  new_number(v);
  if (t < mp_dependent)  number_clone(v, value_number(p));
  set_number_to_zero(mp→max_c[mp_dependent]);
  set_number_to_zero(mp→max_c[mp_proto_dependent]);
  mp→max_link[mp_dependent] = Λ;
  mp→max_link[mp_proto_dependent] = Λ;
  q = (mp_value_node) mp_link(mp→dep_head);
  while (q ≠ mp→dep_head) {
    s = (mp_value_node) mp→temp_head;
    set_mp_link(s, dep_list(q));
    while (1) {
      r = (mp_value_node) mp_link(s);
      if (dep_info(r) ≡ Λ) break;
      if (dep_info(r) ≠ p) {
        s = r;
      }
      else {
        t = mp_type(q);
        if (mp_link(s) ≡ dep_list(q)) {      /* reset the dep_list */
          set_dep_list(q, mp_link(r));
        }
        set_mp_link(s, mp_link(r));
        set_dep_info(r, (mp_node) q);
        number_clone(test, dep_value(r));
        number_abs(test);
        if (number_greater(test, mp→max_c[t])) {      /* Record a new maximum coefficient of type t */
          if (number_positive(mp→max_c[t])) {
            set_mp_link(mp→max_ptr[t], (mp_node) mp→max_link[t]);
            mp→max_link[t] = mp→max_ptr[t];
          }
```

```
            number_clone(mp→max_c[t], test);
            mp→max_ptr[t] = r;
          }
          else {
            set_mp_link(r, (mp_node) mp→max_link[t]);
            mp→max_link[t] = r;
          }
        }
      }
    }
    q = (mp_value_node) mp_link(r);
  }
  if (number_positive(mp→max_c[mp_dependent]) ∨ number_positive(mp→max_c[mp_proto_dependent])) {
      /∗ Choose a dependent variable to take the place of the disappearing independent variable, and
         change all remaining dependencies accordingly ∗/
    mp_number test, ret;      /∗ temporary use ∗/

    new_number(ret);
    new_number(test);
    number_clone(test, mp→max_c[mp_dependent]);
    number_divide_int(test, 4096);
    if (number_greaterequal(test, mp→max_c[mp_proto_dependent])) t = mp_dependent;
    else  t = mp_proto_dependent;
        /∗ Let s = max_ptr[t]. At this point we have value(s) = ±max_c[t], and dep_info(s) points to
           the dependent variable pp of type t from whose dependency list we have removed node s. We
           must reinsert node s into the dependency list, with coefficient −1.0, and with pp as the new
           independent variable. Since pp will have a larger serial number than any other variable, we
           can put node s at the head of the list. ∗/
        /∗ Determine the dependency list s to substitute for the independent variable p ∗/
    s = mp→max_ptr[t];
    pp = (mp_node) dep_info(s);
    number_clone(v, dep_value(s));
    if (t ≡ mp_dependent) {
      set_dep_value(s, fraction_one_t);
    }
    else {
      set_dep_value(s, unity_t);
    }
    number_negate(dep_value(s));
    r = (mp_value_node) dep_list((mp_value_node) pp);
    set_mp_link(s, (mp_node) r);
    while (dep_info(r) ≠ Λ)  r = (mp_value_node) mp_link(r);
    q = (mp_value_node) mp_link(r);
    set_mp_link(r, Λ);
    set_prev_dep(q, prev_dep((mp_value_node) pp));
    set_mp_link(prev_dep((mp_value_node) pp), (mp_node) q);
    mp_new_indep(mp, pp);
    if (cur_exp_node() ≡ pp ∧ mp→cur_exp.type ≡ t)  mp→cur_exp.type = mp_independent;
    if (number_positive(internal_value(mp_tracing_equations))) {
        /∗ Show the transformed dependency ∗/
      if (mp_interesting(mp, p)) {
        mp_begin_diagnostic(mp);
        mp_show_transformed_dependency(mp, v, t, p);
        mp_print_dependency(mp, s, t);
```

```
        mp_end_diagnostic(mp, false);
    }
}
t = (quarterword)(mp_dependent + mp_proto_dependent − t);      /* complement t */
if (number_positive(mp→max_c[t])) {       /* we need to pick up an unchosen dependency */
    set_mp_link(mp→max_ptr[t], (mp_node) mp→max_link[t]);
    mp→max_link[t] = mp→max_ptr[t];
}       /* Finally, there are dependent and proto-dependent variables whose dependency lists must be
          brought up to date. */
if (t ≠ mp_dependent) {       /* Substitute new dependencies in place of p */
    for (t = mp_dependent; t ≤ mp_proto_dependent; t = t + 1) {
        r = mp→max_link[t];
        while (r ≠ Λ) {
            q = (mp_value_node) dep_info(r);
            number_clone(test, v);
            number_negate(test);
            make_fraction(ret, dep_value(r), test);
            set_dep_list(q, mp_p_plus_fq(mp, (mp_value_node) dep_list(q), ret, s, t, mp_dependent));
            if (dep_list(q) ≡ (mp_node) mp→dep_final) mp_make_known(mp, q, mp→dep_final);
            q = r;
            r = (mp_value_node) mp_link(r);
            mp_free_dep_node(mp, q);
        }
    }
}
else {       /* Substitute new proto-dependencies in place of p */
    for (t = mp_dependent; t ≤ mp_proto_dependent; t = t + 1) {
        r = mp→max_link[t];
        while (r ≠ Λ) {
            q = (mp_value_node) dep_info(r);
            if (t ≡ mp_dependent) {       /* for safety's sake, we change q to mp_proto_dependent */
                if (cur_exp_node() ≡ (mp_node) q ∧ mp→cur_exp.type ≡ mp_dependent)
                    mp→cur_exp.type = mp_proto_dependent;
                set_dep_list(q, mp_p_over_v(mp, (mp_value_node) dep_list(q), unity_t, mp_dependent,
                    mp_proto_dependent));
                mp_type(q) = mp_proto_dependent;
                fraction_to_round_scaled(dep_value(r));
            }
            number_clone(test, v);
            number_negate(test);
            make_scaled(ret, dep_value(r), test);
            set_dep_list(q, mp_p_plus_fq(mp, (mp_value_node) dep_list(q), ret, s, mp_proto_dependent,
                mp_proto_dependent));
            if (dep_list(q) ≡ (mp_node) mp→dep_final) mp_make_known(mp, q, mp→dep_final);
            q = r;
            r = (mp_value_node) mp_link(r);
            mp_free_dep_node(mp, q);
        }
    }
}
mp_flush_node_list(mp, (mp_node) s);
if (mp→fix_needed) mp_fix_dependencies(mp);
```

```
        check_arith( );
        free_number(ret);
     }
     free_number(v);
     free_number(test);
  }
```
This code is used in section 922.

924.   ⟨Declarations 8⟩ +≡
  static void *mp_show_transformed_dependency*(**MP** *mp*, **mp_number** *v*, *mp_variable_typet*, **mp_node** *p*);

925.   **static void** *mp_show_transformed_dependency*(**MP** *mp*, **mp_number** *v*, *mp_variable_typet*, **mp_node** *p*)
  {
     mp_number *vv*;   /* for temp use */

     new_number(*vv*);
     mp_print_nl(*mp*, "###␣");
     if (*number_positive*(*v*)) *mp_print_char*(*mp*, *xord*('−'));
     if (*t* ≡ *mp_dependent*) {
        number_clone(*vv*, *mp*→*max_c*[*mp_dependent*]);
        fraction_to_round_scaled(*vv*);
     }
     else {
        number_clone(*vv*, *mp*→*max_c*[*mp_proto_dependent*]);
     }
     if (¬*number_equal*(*vv*, *unity_t*)) {
        print_number(*vv*);
     }
     mp_print_variable_name(*mp*, *p*);
     while (*indep_scale*(*p*) > 0) {
        mp_print(*mp*, "*4");
        set_indep_scale(*p*, *indep_scale*(*p*) − 2);
     }
     if (*t* ≡ *mp_dependent*) *mp_print_char*(*mp*, *xord*('='));
     else *mp_print*(*mp*, "␣=␣");
     free_number(*vv*);
  }

926.   The code for independency removal makes use of three non-symbolic arrays.

⟨Global variables 14⟩ +≡
  mp_number *max_c*[*mp_proto_dependent* + 1];   /* max coefficient magnitude */
  mp_value_node *max_ptr*[*mp_proto_dependent* + 1];   /* where *p* occurs with *max_c* */
  mp_value_node *max_link*[*mp_proto_dependent* + 1];   /* other occurrences of *p* */

927.   ⟨Initialize table entries 182⟩ +≡
  {
     int *i*;
     for (*i* = 0; *i* < *mp_proto_dependent* + 1; *i*++) {
        new_number(*mp*→*max_c*[*i*]);
     }
  }

928.   ⟨Dealloc variables 27⟩ +≡
```
{
  int i;
  for (i = 0;  i < mp_proto_dependent + 1;  i++) {
    free_number(mp→max_c[i]);
  }
}
```

929.   A global variable *var_flag* is set to a special command code just before METAPOST calls *scan_expression*,■
if the expression should be treated as a variable when this command code immediately follows. For example,
var_flag is set to *assignment* at the beginning of a statement, because we want to know the *location* of a
variable at the left of ':=', not the *value* of that variable.

The *scan_expression* subroutine calls *scan_tertiary*, which calls *scan_secondary*, which calls *scan_primary*,
which sets *var_flag*: = 0. In this way each of the scanning routines "knows" when it has been called with a
special *var_flag*, but *var_flag* is usually zero.

A variable preceding a command that equals *var_flag* is converted to a token list rather than a value.
Furthermore, an '=' sign following an expression with *var_flag* = *assignment* is not considered to be a
relation that produces boolean expressions.

⟨Global variables 14⟩ +≡
    int *var_flag*;   /∗ command that wants a variable ∗/

930.   ⟨Set initial values of key variables 38⟩ +≡
    mp→var_flag = 0;

931.   Parsing primary expressions.   The first parsing routine, *scan_primary*, is also the most complicated one, since it involves so many different cases. But each case—with one exception—is fairly simple by itself.

When *scan_primary* begins, the first token of the primary to be scanned should already appear in *cur_cmd*, *cur_mod*, and *cur_sym*. The values of *cur_type* and *cur_exp* should be either dead or dormant, as explained earlier. If *cur_cmd* is not between *min_primary_command* and *max_primary_command*, inclusive, a syntax error will be signaled.

Later we'll come to procedures that perform actual operations like addition, square root, and so on; our purpose now is to do the parsing. But we might as well mention those future procedures now, so that the suspense won't be too bad:

do_nullary(*c*) does primitive operations that have no operands (e.g., '**true**' or '**pencircle**');

do_unary(*c*) applies a primitive operation to the current expression;

do_binary(*p*, *c*) applies a primitive operation to the capsule *p* and the current expression.

⟨ Declare the basic parsing subroutines 931 ⟩ ≡
```
  static void check_for_mediation(MP mp);

  void mp_scan_primary(MP mp)
  {
    mp_command_code my_var_flag;        /* initial value of my_var_flag */
    my_var_flag = mp→var_flag;
    mp→var_flag = 0;
RESTART: check_arith();        /* Supply diagnostic information, if requested */
    if (mp→interrupt ≠ 0) {
      if (mp→OK_to_interrupt) {
        mp_back_input(mp);
        check_interrupt;
        mp_get_x_next(mp);
      }
    }
    switch (cur_cmd()) {
    case mp_left_delimiter:
      {      /* Scan a delimited primary */
        mp_node p, q, r;     /* for list manipulation */
        mp_sym l_delim, r_delim;      /* hash addresses of a delimiter pair */

        l_delim = cur_sym();
        r_delim = equiv_sym(cur_sym());
        mp_get_x_next(mp);
        mp_scan_expression(mp);
        if ((cur_cmd() ≡ mp_comma) ∧ (mp→cur_exp.type ≥ mp_known))
          {      /* Scan the rest of a delimited set of numerics */      /* This code uses the fact that
              red_part and green_part are synonymous with x_part and y_part. */
          p = mp_stash_cur_exp(mp);
          mp_get_x_next(mp);
          mp_scan_expression(mp);
            /* Make sure the second part of a pair or color has a numeric type */
          if (mp→cur_exp.type < mp_known) {
            const char *hlp[] = {"I've␣started␣to␣scan␣a␣pair␣'(a,b)'␣or␣a␣color␣'(a,b,c)';",
                "but␣after␣finding␣a␣nice␣'a'␣I␣found␣a␣'b'␣that␣isn't",
                "of␣numeric␣type.␣So␣I've␣changed␣that␣part␣to␣zero.",
                "(The␣b␣that␣I␣didn't␣like␣appears␣above␣the␣error␣message.)", Λ};
            mp_value new_expr;
```

```
            memset(&new_expr, 0, sizeof(mp_value));
            mp_disp_err(mp, Λ);
            new_number(new_expr.data.n);
            set_number_to_zero(new_expr.data.n);
            mp_back_error(mp, "Nonnumeric␣ypart␣has␣been␣replaced␣by␣0", hlp, true);
            mp_get_x_next(mp);
            mp_flush_cur_exp(mp, new_expr);
        }
        q = mp_get_value_node(mp);
        mp_name_type(q) = mp_capsule;
        if (cur_cmd( ) ≡ mp_comma) {
            mp_init_color_node(mp, q);
            r = value_node(q);
            mp_stash_in(mp, y_part(r));
            mp_unstash_cur_exp(mp, p);
            mp_stash_in(mp, x_part(r));          /* Scan the last of a triplet of numerics */
            mp_get_x_next(mp);
            mp_scan_expression(mp);
            if (mp⃗cur_exp.type < mp_known) {
                mp_value new_expr;
                const char *hlp[ ] = {"I'␣just␣scanned␣a␣color␣'(a,b,c)'␣or\
                    cmykcolor(a,b,c,d);␣but␣the␣'c'",
                    "isn't␣of␣numeric␣type.␣So␣I've␣changed␣that␣part␣to␣zero.",
                    "(The␣c␣that␣I␣didn't␣like␣appears␣above␣the␣error␣message.)", Λ};

                memset(&new_expr, 0, sizeof(mp_value));
                mp_disp_err(mp, Λ);
                new_number(new_expr.data.n);
                set_number_to_zero(new_expr.data.n);
                mp_back_error(mp, "Nonnumeric␣third␣part␣has␣been␣replaced␣by␣0", hlp, true);
                mp_get_x_next(mp);
                mp_flush_cur_exp(mp, new_expr);
            }
            mp_stash_in(mp, blue_part(r));
            if (cur_cmd( ) ≡ mp_comma) {
                mp_node t;       /* a token */

                mp_init_cmykcolor_node(mp, q);
                t = value_node(q);
                mp_type(cyan_part(t)) = mp_type(red_part(r));
                set_value_number(cyan_part(t), value_number(red_part(r)));
                mp_type(magenta_part(t)) = mp_type(green_part(r));
                set_value_number(magenta_part(t), value_number(green_part(r)));
                mp_type(yellow_part(t)) = mp_type(blue_part(r));
                set_value_number(yellow_part(t), value_number(blue_part(r)));
                mp_recycle_value(mp, r);
                r = t;       /* Scan the last of a quartet of numerics */
                mp_get_x_next(mp);
                mp_scan_expression(mp);
                if (mp⃗cur_exp.type < mp_known) {
                    const char *hlp[ ] = {"I've␣just␣scanned␣a␣cmykcolor␣'(c,m,y,k\
                        )';␣but␣the␣'k'␣isn't",
                        "of␣numeric␣type.␣So␣I've␣changed␣that␣part␣to␣zero.",
                        "(The␣k␣that␣I␣didn't␣like␣appears␣above␣the␣error␣message.)", Λ};
```

```
                    mp_value new_expr;

                    memset(&new_expr, 0, sizeof(mp_value));
                    new_number(new_expr.data.n);
                    mp_disp_err(mp, Λ);
                    set_number_to_zero(new_expr.data.n);
                    mp_back_error(mp, "Nonnumeric␣blackpart␣has␣been␣replaced␣by␣0", hlp, true);
                    mp_get_x_next(mp);
                    mp_flush_cur_exp(mp, new_expr);
                  }
                  mp_stash_in(mp, black_part(r));
                }
              }
              else {
                mp_init_pair_node(mp, q);
                r = value_node(q);
                mp_stash_in(mp, y_part(r));
                mp_unstash_cur_exp(mp, p);
                mp_stash_in(mp, x_part(r));
              }
              mp_check_delimiter(mp, l_delim, r_delim);
              mp→cur_exp.type = mp_type(q);
              set_cur_exp_node(q);
            }
            else {
              mp_check_delimiter(mp, l_delim, r_delim);
            }
          }
        break;
      case mp_begin_group:      /* Scan a grouped primary */
          /* The local variable group_line keeps track of the line where a begingroup command occurred;
             this will be useful in an error message if the group doesn't actually end. */
        {
          integer group_line;      /* where a group began */

          group_line = mp_true_line(mp);
          if (number_positive(internal_value(mp_tracing_commands))) show_cur_cmd_mod;
          mp_save_boundary(mp);
          do {
            mp_do_statement(mp);      /* ends with cur_cmd ≥ semicolon */
          } while (cur_cmd() ≡ mp_semicolon);
          if (cur_cmd() ≠ mp_end_group) {
            char msg[256];
            const char *hlp[] = {"I␣saw␣a␣'begingroup'␣back␣there␣that␣hasn't␣been␣matched",
                "by␣'endgroup'.␣So␣I've␣inserted␣'endgroup'␣now.", Λ};

            mp_snprintf(msg, 256, "A␣group␣begun␣on␣line␣%d␣never␣ended", (int) group_line);
            mp_back_error(mp, msg, hlp, true);
            set_cur_cmd((mp_variable_type)mp_end_group);
          }
          mp_unsave(mp);      /* this might change cur_type, if independent variables are recycled */
          if (number_positive(internal_value(mp_tracing_commands))) show_cur_cmd_mod;
        }
        break;
```

case *mp_string_token*:   /∗ Scan a string constant ∗/
  mp→cur_exp.type = mp_string_type;
  set_cur_exp_str(*cur_mod_str*( ));
  break;
case *mp_numeric_token*:
  {   /∗ Scan a primary that starts with a numeric token ∗/
    /∗ A numeric token might be a primary by itself, or it might be the numerator of a fraction
      composed solely of numeric tokens, or it might multiply the primary that follows (provided that
      the primary doesn't begin with a plus sign or a minus sign). The code here uses the facts that
      max_primary_command = plus_or_minus and *max_primary_command − 1 = numeric_token*. If
      a fraction is found that is less than unity, we try to retain higher precision when we use it in
      scalar multiplication. ∗/
  mp_number *num*, *denom*;   /∗ for primaries that are fractions, like '1/2' ∗/

  new_number(*num*);
  new_number(*denom*);
  set_cur_exp_value_number(*cur_mod_number*( ));
  mp→cur_exp.type = mp_known;
  mp_get_x_next(*mp*);
  if (*cur_cmd*( ) ≠ *mp_slash*) {
    set_number_to_zero(*num*);
    set_number_to_zero(*denom*);
  }
  else {
    mp_get_x_next(*mp*);
    if (*cur_cmd*( ) ≠ *mp_numeric_token*) {
      mp_back_input(*mp*);
      set_cur_cmd((*mp_variable_type*)*mp_slash*);
      set_cur_mod(*mp_over*);
      set_cur_sym(*mp→frozen_slash*);
      free_number(*num*);
      free_number(*denom*);
      goto DONE;
    }
    number_clone(*num*, *cur_exp_value_number*( ));
    number_clone(*denom*, *cur_mod_number*( ));
    if (*number_zero*(*denom*)) {   /∗ Protest division by zero ∗/
      const char ∗*hlp*[ ] = {"I'll␣pretend␣that␣you␣meant␣to␣divide␣by␣1.", Λ};
      mp_error(*mp*, "Division␣by␣zero", *hlp*, *true*);
    }
    else {
      mp_number *ret*;
      new_number(*ret*);
      make_scaled(*ret*, *num*, *denom*);
      set_cur_exp_value_number(*ret*);
      free_number(*ret*);
    }
    check_arith( );
    mp_get_x_next(*mp*);
  }
  if (*cur_cmd*( ) ≥ *mp_min_primary_command*) {
    if (*cur_cmd*( ) < *mp_numeric_token*) {   /∗ in particular, *cur_cmd <> plus_or_minus* ∗/

```
          mp_node p;        /* for list manipulation */
          mp_number absnum, absdenom;

          new_number(absnum);
          new_number(absdenom);
          p = mp_stash_cur_exp(mp);
          mp_scan_primary(mp);
          number_clone(absnum, num);
          number_abs(absnum);
          number_clone(absdenom, denom);
          number_abs(absdenom);
          if (number_greaterequal(absnum, absdenom) ∨ (mp→cur_exp.type < mp_color_type)) {
            mp_do_binary(mp, p, mp_times);
          }
          else {
            mp_frac_mult(mp, num, denom);
            mp_free_value_node(mp, p);
          }
          free_number(absnum);
          free_number(absdenom);
        }
      }
      free_number(num);
      free_number(denom);
      goto DONE;
    }
    break;
  case mp_nullary:        /* Scan a nullary operation */
    mp_do_nullary(mp, (quarterword) cur_mod());
    break;
  case mp_unary: case mp_type_name: case mp_cycle: case mp_plus_or_minus:
    {      /* Scan a unary operation */
      quarterword c;        /* a primitive operation code */

      c = (quarterword) cur_mod();
      mp_get_x_next(mp);
      mp_scan_primary(mp);
      mp_do_unary(mp, c);
      goto DONE;
    }
    break;
  case mp_primary_binary:
    {      /* Scan a binary operation with 'of' between its operands */
      mp_node p;        /* for list manipulation */
      quarterword c;        /* a primitive operation code */

      c = (quarterword) cur_mod();
      mp_get_x_next(mp);
      mp_scan_expression(mp);
      if (cur_cmd() ≠ mp_of_token) {
        char msg[256];
        mp_string sname;
        const char *hlp[] = {"I've␣got␣the␣first␣argument;␣will␣look␣now␣for␣the␣other.", Λ};
        int old_setting = mp→selector;
```

```
        mp→selector = new_string;
        mp_print_cmd_mod(mp, mp_primary_binary, c);
        mp→selector = old_setting;
        sname = mp_make_string(mp);
        mp_snprintf(msg, 256, "Missing␣'of'␣has␣been␣inserted␣for␣%s", mp_str(mp, sname));
        delete_str_ref(sname);
        mp_back_error(mp, msg, hlp, true);
      }
      p = mp_stash_cur_exp(mp);
      mp_get_x_next(mp);
      mp_scan_primary(mp);
      mp_do_binary(mp, p, c);
      goto DONE;
    }
    break;
  case mp_str_op:      /* Convert a suffix to a string */
    mp_get_x_next(mp);
    mp_scan_suffix(mp);
    mp→old_setting = mp→selector;
    mp→selector = new_string;
    mp_show_token_list(mp, cur_exp_node( ), Λ, 100000, 0);
    mp_flush_token_list(mp, cur_exp_node( ));
    set_cur_exp_str(mp_make_string(mp));
    mp→selector = mp→old_setting;
    mp→cur_exp.type = mp_string_type;
    goto DONE;
    break;
  case mp_internal_quantity:      /* Scan an internal numeric quantity */      /* If an internal quantity
        appears all by itself on the left of an assignment, we return a token list of length one, containing
        the address of the internal quantity, with name_type equal to mp_internal_sym. (This accords
        with the conventions of the save stack, as described earlier.) */
    {
      halfword qq = cur_mod( );
      if (my_var_flag ≡ mp_assignment) {
        mp_get_x_next(mp);
        if (cur_cmd( ) ≡ mp_assignment) {
          set_cur_exp_node(mp_get_symbolic_node(mp));
          set_mp_sym_info(cur_exp_node( ), qq);
          mp_name_type(cur_exp_node( )) = mp_internal_sym;
          mp→cur_exp.type = mp_token_list;
          goto DONE;
        }
        mp_back_input(mp);
      }
      if (internal_type(qq) ≡ mp_string_type) {
        set_cur_exp_str(internal_string(qq));
      }
      else {
        set_cur_exp_value_number(internal_value(qq));
      }
      mp→cur_exp.type = internal_type(qq);
    }
```

      break;
    case $mp\_capsule\_token$: $mp\_make\_exp\_copy(mp, cur\_mod\_node(\,))$;
      break;
    case mp_tag_token: ⟨ Scan a variable primary; **goto** $restart$ if it turns out to be a macro 936 ⟩;
      break;
    default: $mp\_bad\_exp(mp, \texttt{"A}_\sqcup\texttt{primary"})$;
      goto RESTART;
      break;
    }
    $mp\_get\_x\_next(mp)$;   /∗ the routines **goto** $done$ if they don't want this ∗/
  DONE: $check\_for\_mediation(mp)$;
  }

See also sections 932, 943, 944, 946, 947, 948, and 953.

This code is used in section 1285.

932.   Expressions of the form 'a[b,c]' are converted into 'b+a*(c-b)', without checking the types of
b or c, provided that a is numeric.

⟨Declare the basic parsing subroutines 931⟩ +≡

```
static void check_for_mediation(MP mp)
{
   mp_node p, q, r;      /* for list manipulation */
   if (cur_cmd() ≡ mp_left_bracket) {
     if (mp→cur_exp.type ≥ mp_known) {       /* Scan a mediation construction */
       p = mp_stash_cur_exp(mp);
       mp_get_x_next(mp);
       mp_scan_expression(mp);
       if (cur_cmd() ≠ mp_comma) {       /* Put the left bracket and the expression back to be
                rescanned */       /* The left bracket that we thought was introducing a subscript might
                have actually been the left bracket in a mediation construction like 'x[a,b]'. So we don't
                issue an error message at this point; but we do want to back up so as to avoid any
                embarrassment about our incorrect assumption. */
         mp_back_input(mp);       /* that was the token following the current expression */
         mp_back_expr(mp);
         set_cur_cmd((mp_variable_type)mp_left_bracket);
         set_cur_mod_number(zero_t);
         set_cur_sym(mp→frozen_left_bracket);
         mp_unstash_cur_exp(mp, p);
       }
       else {
         q = mp_stash_cur_exp(mp);
         mp_get_x_next(mp);
         mp_scan_expression(mp);
         if (cur_cmd() ≠ mp_right_bracket) {
           const char *hlp[] = {"I've␣scanned␣an␣expression␣of␣the␣form␣'a[b,c',",
               "so␣a␣right␣bracket␣should␣have␣come␣next.",
               "I␣shall␣pretend␣that␣one␣was␣there.", Λ};
           mp_back_error(mp, "Missing␣']'␣has␣been␣inserted", hlp, true);
         }
         r = mp_stash_cur_exp(mp);
         mp_make_exp_copy(mp, q);
         mp_do_binary(mp, r, mp_minus);
         mp_do_binary(mp, p, mp_times);
         mp_do_binary(mp, q, mp_plus);
         mp_get_x_next(mp);
       }
     }
   }
}
```

933. Errors at the beginning of expressions are flagged by *bad_exp*.

```
static void mp_bad_exp(MP mp, const char *s)
{
  char msg[256];
  int save_flag;
  const char *hlp[] = {"I'm␣afraid␣I␣need␣some␣sort␣of␣value␣in␣order␣to␣continue,",
      "so␣I've␣tentatively␣inserted␣'0'.␣You␣may␣want␣to",
      "delete␣this␣zero␣and␣insert␣something␣else;",
      "see␣Chapter␣27␣of␣The␣METAFONTbook␣for␣an␣example.", Λ};
  ;
  {
    mp_string cm;
    int old_selector = mp→selector;
    mp→selector = new_string;
    mp_print_cmd_mod(mp, cur_cmd( ), cur_mod( ));
    mp→selector = old_selector;
    cm = mp_make_string(mp);
    mp_snprintf(msg, 256, "%s␣expression␣can't␣begin␣with␣'%s'", s, mp_str(mp, cm));
    delete_str_ref(cm);
  }
  mp_back_input(mp);
  set_cur_sym(Λ);
  set_cur_cmd((mp_variable_type) mp_numeric_token);
  set_cur_mod_number(zero_t);
  mp_ins_error(mp, msg, hlp, true);
  save_flag = mp→var_flag;
  mp→var_flag = 0;
  mp_get_x_next(mp);
  mp→var_flag = save_flag;
}
```

934.   The *stash_in* subroutine puts the current (numeric) expression into a field within a "big node."

static void $mp\_stash\_in(\mathbf{MP}\ mp, \mathbf{mp\_node}\ p)$
{
    mp_value_node q;   /∗ temporary register ∗/
    $mp\_type(p) = mp\text{-}cur\_exp.type$;
    if $(mp\text{-}cur\_exp.type \equiv mp\_known)$ {
      $set\_value\_number(p, cur\_exp\_value\_number())$;
    }
    else {
      if $(mp\text{-}cur\_exp.type \equiv mp\_independent)$ {   /∗ Stash an independent *cur_exp* into a big node
        ∗/   /∗ In rare cases the current expression can become *independent*. There may be many
        dependency lists pointing to such an independent capsule, so we can't simply move it into
        place within a big node. Instead, we copy it, then recycle it. ∗/
      $q = mp\_single\_dependency(mp, cur\_exp\_node())$;
      if $(q \equiv mp\text{-}dep\_final)$ {
        $mp\_type(p) = mp\_known$;
        $set\_value\_number(p, zero\_t)$;
        $mp\_free\_dep\_node(mp, q)$;
      }
      else {
        $mp\_new\_dep(mp, p, mp\_dependent, q)$;
      }
      $mp\_recycle\_value(mp, cur\_exp\_node())$;
      $mp\_free\_value\_node(mp, cur\_exp\_node())$;
    }
    else {
      $set\_dep\_list((\mathbf{mp\_value\_node})\ p, dep\_list((\mathbf{mp\_value\_node})\ cur\_exp\_node()))$;
      $set\_prev\_dep((\mathbf{mp\_value\_node})\ p, prev\_dep((\mathbf{mp\_value\_node})\ cur\_exp\_node()))$;
      $set\_mp\_link(prev\_dep((\mathbf{mp\_value\_node})\ p), p)$;
      $mp\_free\_dep\_node(mp, (\mathbf{mp\_value\_node})\ cur\_exp\_node())$;
    }
    }
    $mp\text{-}cur\_exp.type = mp\_vacuous$;
}

935.   The most difficult part of *scan_primary* has been saved for last, since it was necessary to build up some confidence first. We can now face the task of scanning a variable.

As we scan a variable, we build a token list containing the relevant names and subscript values, simultaneously following along in the "collective" structure to see if we are actually dealing with a macro instead of a value.

The local variables *pre_head* and *post_head* will point to the beginning of the prefix and suffix lists; *tail* will point to the end of the list that is currently growing.

Another local variable, *tt*, contains partial information about the declared type of the variable-so-far. If $tt \geq mp\_unsuffixed\_macro$, the relation $tt = mp\_type(q)$ will always hold. If $tt = undefined$, the routine doesn't bother to update its information about type. And if $undefined < tt < mp\_unsuffixed\_macro$, the precise value of tt isn't critical.

936.   ⟨Scan a variable primary; **goto** *restart* if it turns out to be a macro 936⟩ ≡

{
   mp_node *p*, *q*;   /∗ for list manipulation ∗/
   mp_node *t*;   /∗ a token ∗/
   mp_node *pre_head*, *post_head*, *tail*;   /∗ prefix and suffix list variables ∗/
   quarterword *tt*;   /∗ approximation to the type of the variable-so-far ∗/
   mp_node *macro_ref* = 0;   /∗ reference count for a suffixed macro ∗/
   pre_head = *mp_get_symbolic_node*(*mp*);
   tail = *pre_head*;
   post_head = Λ;
   tt = *mp_vacuous*;
   while (1) {
    t = *mp_cur_tok*(*mp*);
    mp_link(*tail*) = *t*;
    if (*tt* ≠ *mp_undefined*) {   /∗ Find the approximate type *tt* and corresponding *q* ∗/
      /∗ Every time we call *get_x_next*, there's a chance that the variable we've been looking at will
        disappear. Thus, we cannot safely keep *q* pointing into the variable structure; we need to
        start searching from the root each time. ∗/
     mp_sym *qq*;
     p = *mp_link*(*pre_head*);
     qq = *mp_sym_sym*(*p*);
     tt = *mp_undefined*;
     if (*eq_type*(*qq*) % *mp_outer_tag* ≡ *mp_tag_token*) {
      q = *equiv_node*(*qq*);
      if (*q* ≡ Λ) **goto** DONE2;
      while (1) {
       p = *mp_link*(*p*);
       if (*p* ≡ Λ) {
        tt = *mp_type*(*q*);
        goto DONE2;
       }
       if (*mp_type*(*q*) ≠ *mp_structured*) **goto** DONE2;
       q = *mp_link*(*attr_head*(*q*));   /∗ the *collective_subscript* attribute ∗/
       if (*mp_type*(*p*) ≡ *mp_symbol_node*) {   /∗ it's not a subscript ∗/
        do {
         q = *mp_link*(*q*);
        } **while** (¬(*hashloc*(*q*) ≥ *mp_sym_sym*(*p*)));
        if (*hashloc*(*q*) > *mp_sym_sym*(*p*)) **goto** DONE2;
       }
      }
     }
    }
    DONE2:
    if (*tt* ≥ *mp_unsuffixed_macro*) {
     /∗ Either begin an unsuffixed macro call or prepare for a suffixed one ∗/
     mp_link(*tail*) = Λ;
     if (*tt* > *mp_unsuffixed_macro*) {   /∗ *tt* = *mp_suffixed_macro* ∗/
      post_head = *mp_get_symbolic_node*(*mp*);
      tail = *post_head*;
      mp_link(*tail*) = *t*;
      tt = *mp_undefined*;
      macro_ref = *value_node*(*q*);
      add_mac_ref(*macro_ref*);

```
      }
      else {      /* Set up unsuffixed macro call and goto restart */      /* The only complication
              associated with macro calling is that the prefix and "at" parameters must be packaged in
              an appropriate list of lists. */
        p = mp_get_symbolic_node(mp);
        set_mp_sym_sym(pre_head, mp_link(pre_head));
        mp_link(pre_head) = p;
        set_mp_sym_sym(p, t);
        mp_macro_call(mp, value_node(q), pre_head, Λ);
        mp_get_x_next(mp);
        goto RESTART;
      }
    }
  }
  mp_get_x_next(mp);
  tail = t;
  if (cur_cmd() ≡ mp_left_bracket) {
      /* Scan for a subscript; replace cur_cmd by numeric_token if found */
    mp_get_x_next(mp);
    mp_scan_expression(mp);
    if (cur_cmd() ≠ mp_right_bracket) {      /* Put the left bracket and the expression back to
              be rescanned */      /* The left bracket that we thought was introducing a subscript
              might have actually been the left bracket in a mediation construction like 'x[a,b]'. So we
              don't issue an error message at this point; but we do want to back up so as to avoid any
              embarrassment about our incorrect assumption. */
      mp_back_input(mp);      /* that was the token following the current expression */
      mp_back_expr(mp);
      set_cur_cmd((mp_variable_type)mp_left_bracket);
      set_cur_mod_number(zero_t);
      set_cur_sym(mp→frozen_left_bracket);
    }
    else {
      if (mp→cur_exp.type ≠ mp_known)  mp_bad_subscript(mp);
      set_cur_cmd((mp_variable_type)mp_numeric_token);
      set_cur_mod_number(cur_exp_value_number());
      set_cur_sym(Λ);
    }
  }
  if (cur_cmd() > mp_max_suffix_token) break;
  if (cur_cmd() < mp_min_suffix_token) break;
}    /* now cur_cmd is internal_quantity, tag_token, or numeric_token */      /* Handle unusual
        cases that masquerade as variables, and goto restart or goto done if appropriate; otherwise
        make a copy of the variable and goto done */
    /* If the variable does exist, we also need to check for a few other special cases before deciding
        that a plain old ordinary variable has, indeed, been scanned. */
if (post_head ≠ Λ) {      /* Set up suffixed macro call and goto restart */
    /* If the "variable" that turned out to be a suffixed macro no longer exists, we don't care, because
        we have reserved a pointer (macro_ref) to its token list. */
  mp_back_input(mp);
  p = mp_get_symbolic_node(mp);
  q = mp_link(post_head);
  set_mp_sym_sym(pre_head, mp_link(pre_head));
```

```
        mp_link(pre_head) = post_head;
        set_mp_sym_sym(post_head, q);
        mp_link(post_head) = p;
        set_mp_sym_sym(p, mp_link(q));
        mp_link(q) = Λ;
        mp_macro_call(mp, macro_ref, pre_head, Λ);
        decr_mac_ref(macro_ref);
        mp_get_x_next(mp);
        goto RESTART;
    }
    q = mp_link(pre_head);
    mp_free_symbolic_node(mp, pre_head);
    if (cur_cmd() ≡ my_var_flag) {
        mp→cur_exp.type = mp_token_list;
        set_cur_exp_node(q);
        goto DONE;
    }
    p = mp_find_variable(mp, q);
    if (p ≠ Λ) {
        mp_make_exp_copy(mp, p);
    }
    else {
        mp_value new_expr;
        const char *hlp[] = {"While␣I␣was␣evaluating␣the␣suffix␣of␣this␣variable,",
            "something␣was␣redefined,␣and␣it's␣no␣longer␣a␣variable!",
            "In␣order␣to␣get␣back␣on␣my␣feet,␣I've␣inserted␣'0'␣instead.", Λ};
        char *msg = mp_obliterated(mp, q);

        memset(&new_expr, 0, sizeof(mp_value));
        new_number(new_expr.data.n);
        set_number_to_zero(new_expr.data.n);
        mp_back_error(mp, msg, hlp, true);
        free(msg);
        mp_get_x_next(mp);
        mp_flush_cur_exp(mp, new_expr);
    }
    mp_flush_node_list(mp, q);
    goto DONE;
}
```

This code is used in section 931.

937.   Here's a routine that puts the current expression back to be read again.

```
static void mp_back_expr(MP mp)
{
    mp_node p;      /* capsule token */
    p = mp_stash_cur_exp(mp);
    mp_link(p) = Λ;
    back_list(p);
}
```

938.   Unknown subscripts lead to the following error message.

static void $mp\_bad\_subscript(\mathbf{MP}\ mp)$
{
　$\mathbf{mp\_value}\ new\_expr;$
　$\mathbf{const\ char}\ *hlp[\,] = \{$"A␣bracketed␣subscript␣must␣have␣a␣known␣numeric␣value;",
　　　"unfortunately,␣what␣I␣found␣was␣the␣value␣that␣appears␣just",
　　　"above␣this␣error␣message.␣So␣I'll␣try␣a␣zero␣subscript.", $\Lambda\};$
　$memset(\&new\_expr, 0, \mathbf{sizeof}(\mathbf{mp\_value}));$
　$new\_number(new\_expr.data.n);$
　$mp\_disp\_err(mp, \Lambda);$
　$mp\_error(mp,$ "Improper␣subscript␣has␣been␣replaced␣by␣zero", $hlp, true);$
　;
　$mp\_flush\_cur\_exp(mp, new\_expr);$
}

939.   How do things stand now? Well, we have scanned an entire variable name, including possible subscripts and/or attributes; cur_cmd, cur_mod, and cur_sym represent the token that follows. If $post\_head = \Lambda$, a token list for this variable name starts at $mp\_link(pre\_head)$, with all subscripts evaluated. But if $post\_head <> \Lambda$, the variable turned out to be a suffixed macro; pre_head is the head of the prefix list, while $post\_head$ is the head of a token list containing both '@!' and the suffix.

Our immediate problem is to see if this variable still exists. (Variable structures can change drastically whenever we call get_x_next; users aren't supposed to do this, but the fact that it is possible means that we must be cautious.)

The following procedure creates an error message for when a variable unexpectedly disappears.

static char $*mp\_obliterated(\mathbf{MP}\ mp, \mathbf{mp\_node}\ q)$
{
　$\mathbf{char}\ msg[256];$
　$\mathbf{mp\_string}\ sname;$
　$\mathbf{int}\ old\_setting = mp\text{-}selector;$
　$mp\text{-}selector = new\_string;$
　$mp\_show\_token\_list(mp, q, \Lambda, 1000, 0);$
　$sname = mp\_make\_string(mp);$
　$mp\text{-}selector = old\_setting;$
　$mp\_snprintf(msg, 256,$ "Variable␣%s␣has␣been␣obliterated", $mp\_str(mp, sname));$
　;
　$delete\_str\_ref(sname);$
　return $xstrdup(msg);$
}

940.   Our remaining job is simply to make a copy of the value that has been found. Some cases are harder than others, but complexity arises solely because of the multiplicity of possible cases.

⟨ Declare the procedure called *make_exp_copy* 940 ⟩ ≡
  ⟨ Declare subroutines needed by *make_exp_copy* 941 ⟩;

  static void *mp_make_exp_copy*(**MP** *mp*, **mp_node** *p*)
  {
    mp_node *t*;   /* register(s) for list manipulation */
    mp_value_node *q*;
  RESTART: *mp→cur_exp.type* = *mp_type*(*p*);
    switch (*mp→cur_exp.type*) {
    case *mp_vacuous*: **case** *mp_boolean_type*: **case** *mp_known*:
      set_cur_exp_value_number(*value_number*(*p*));
      break;
    case *unknown_types*: *t* = *mp_new_ring_entry*(*mp*, *p*);
      set_cur_exp_node(*t*);
      break;
    case *mp_string_type*: *set_cur_exp_str*(*value_str*(*p*));
      break;
    case *mp_picture_type*: *set_cur_exp_node*(*value_node*(*p*));
      add_edge_ref(*cur_exp_node*( ));
      break;
    case *mp_pen_type*: *set_cur_exp_knot*(*copy_pen*(*value_knot*(*p*)));
      break;
    case *mp_path_type*: *set_cur_exp_knot*(*mp_copy_path*(*mp*, *value_knot*(*p*)));
      break;
    case *mp_transform_type*: **case** *mp_color_type*: **case** *mp_cmykcolor_type*: **case** *mp_pair_type*:
      /* Copy the big node *p* */   /* The most tedious case arises when the user refers to a **pair**,
        color, or **transform** variable; we must copy several fields, each of which can be *independent*,
        dependent, *mp_proto_dependent*, or *known*. */
      if (*value_node*(*p*) ≡ Λ) {
        switch (*mp_type*(*p*)) {
        case *mp_pair_type*: *mp_init_pair_node*(*mp*, *p*);
          break;
        case *mp_color_type*: *mp_init_color_node*(*mp*, *p*);
          break;
        case *mp_cmykcolor_type*: *mp_init_cmykcolor_node*(*mp*, *p*);
          break;
        case *mp_transform_type*: *mp_init_transform_node*(*mp*, *p*);
          break;
        default: /* there are no other valid cases, but please the compiler */
          break;
        }
      }
      t = *mp_get_value_node*(*mp*);
      mp_name_type(*t*) = *mp_capsule*;
      q = (**mp_value_node**) *value_node*(*p*);
      switch (*mp→cur_exp.type*) {
      case *mp_pair_type*: *mp_init_pair_node*(*mp*, *t*);
        mp_install(*mp*, *y_part*(*value_node*(*t*)), *y_part*(*q*));
        mp_install(*mp*, *x_part*(*value_node*(*t*)), *x_part*(*q*));
        break;

```
        case mp_color_type: mp_init_color_node(mp, t);
          mp_install(mp, blue_part(value_node(t)), blue_part(q));
          mp_install(mp, green_part(value_node(t)), green_part(q));
          mp_install(mp, red_part(value_node(t)), red_part(q));
          break;
        case mp_cmykcolor_type: mp_init_cmykcolor_node(mp, t);
          mp_install(mp, black_part(value_node(t)), black_part(q));
          mp_install(mp, yellow_part(value_node(t)), yellow_part(q));
          mp_install(mp, magenta_part(value_node(t)), magenta_part(q));
          mp_install(mp, cyan_part(value_node(t)), cyan_part(q));
          break;
        case mp_transform_type: mp_init_transform_node(mp, t);
          mp_install(mp, yy_part(value_node(t)), yy_part(q));
          mp_install(mp, yx_part(value_node(t)), yx_part(q));
          mp_install(mp, xy_part(value_node(t)), xy_part(q));
          mp_install(mp, xx_part(value_node(t)), xx_part(q));
          mp_install(mp, ty_part(value_node(t)), ty_part(q));
          mp_install(mp, tx_part(value_node(t)), tx_part(q));
          break;
        default:      /* there are no other valid cases, but please the compiler */
          break;
        }
        set_cur_exp_node(t);
        break;
      case mp_dependent: case mp_proto_dependent:
        mp_encapsulate(mp, mp_copy_dep_list(mp, (mp_value_node) dep_list((mp_value_node) p)));
        break;
      case mp_numeric_type: mp_new_indep(mp, p);
        goto RESTART;
        break;
      case mp_independent: q = mp_single_dependency(mp, p);
        if (q ≡ mp→dep_final) {
          mp→cur_exp.type = mp_known;
          set_cur_exp_value_number(zero_t);
          mp_free_dep_node(mp, q);
        }
        else {
          mp→cur_exp.type = mp_dependent;
          mp_encapsulate(mp, q);
        }
        break;
      default: mp_confusion(mp, "copy");
        ;
        break;
      }
    }
```

This code is used in section 707.

941.   The *encapsulate* subroutine assumes that *dep_final* is the tail of dependency list p.

⟨ Declare subroutines needed by *make_exp_copy* 941 ⟩ ≡
  static void *mp_encapsulate*(**MP** *mp*, **mp_value_node** *p*)
  {
    mp_node *q* = *mp_get_value_node*(*mp*);
    FUNCTION_TRACE2("mp_encapsulate(%p)\n", *p*);
    mp_name_type(*q*) = *mp_capsule*;
    mp_new_dep(*mp*, *q*, *mp*→*cur_exp*.*type*, *p*);
    set_cur_exp_node(*q*);
  }

See also section 942.

This code is used in section 940.

942.   The *install* procedure copies a numeric field q into field r of a big node that will be part of a capsule.

⟨ Declare subroutines needed by *make_exp_copy* 941 ⟩ +≡
  static void *mp_install*(**MP** *mp*, **mp_node** *r*, **mp_node** *q*)
  {
    mp_value_node *p*;   /∗ temporary register ∗/
    if (*mp_type*(*q*) ≡ *mp_known*) {
      mp_type(*r*) = *mp_known*;
      set_value_number(*r*, *value_number*(*q*));
    }
    else if (*mp_type*(*q*) ≡ *mp_independent*) {
      p = *mp_single_dependency*(*mp*, *q*);
      if (*p* ≡ *mp*→*dep_final*) {
        mp_type(*r*) = *mp_known*;
        set_value_number(*r*, *zero_t*);
        mp_free_dep_node(*mp*, *p*);
      }
      else {
        mp_new_dep(*mp*, *r*, *mp_dependent*, *p*);
      }
    }
    else {
      mp_new_dep(*mp*, *r*, *mp_type*(*q*), *mp_copy_dep_list*(*mp*, (**mp_value_node**) *dep_list*((**mp_value_node**)
        q)));
    }
  }

943.   Here is a comparatively simple routine that is used to scan the **suffix** parameters of a macro.

⟨ Declare the basic parsing subroutines 931 ⟩ +≡

```
static void mp_scan_suffix (MP mp)
{
    mp_node h, t;        /∗ head and tail of the list being built ∗/
    mp_node p;        /∗ temporary register ∗/
    h = mp_get_symbolic_node (mp);
    t = h;
    while (1) {
        if (cur_cmd ( ) ≡ mp_left_bracket) {
            /∗ Scan a bracketed subscript and set cur_cmd: = numeric_token ∗/
            mp_get_x_next (mp);
            mp_scan_expression (mp);
            if (mp→cur_exp.type ≠ mp_known)  mp_bad_subscript (mp);
            if (cur_cmd ( ) ≠ mp_right_bracket) {
                const char ∗hlp[ ] = {"I've␣seen␣a␣`['␣and␣a␣subscript␣value,␣in␣a␣suffix,",
                    "so␣a␣right␣bracket␣should␣have␣come␣next.",
                    "I␣shall␣pretend␣that␣one␣was␣there.", Λ};

                mp_back_error (mp, "Missing␣`]'␣has␣been␣inserted", hlp, true);
            }
            set_cur_cmd ((mp_variable_type )mp_numeric_token );
            set_cur_mod_number (cur_exp_value_number ( ));
        }
        if (cur_cmd ( ) ≡ mp_numeric_token) {
            mp_number arg1 ;

            new_number (arg1 );
            number_clone (arg1 , cur_mod_number ( ));
            p = mp_new_num_tok (mp, arg1 );
            free_number (arg1 );
        }
        else if ((cur_cmd ( ) ≡ mp_tag_token) ∨ (cur_cmd ( ) ≡ mp_internal_quantity)) {
            p = mp_get_symbolic_node (mp);
            set_mp_sym_sym (p, cur_sym ( ));
            mp_name_type (p) = cur_sym_mod ( );
        }
        else {
            break;
        }
        mp_link (t) = p;
        t = p;
        mp_get_x_next (mp);
    }
    set_cur_exp_node (mp_link (h));
    mp_free_symbolic_node (mp, h);
    mp→cur_exp.type = mp_token_list ;
}
```

944.   Parsing secondary and higher expressions.
After the intricacies of *scan_primary*, the *scan_secondary* routine is refreshingly simple. It's not trivial, but the operations are relatively straightforward; the main difficulty is, again, that expressions and data structures might change drastically every time we call *get_x_next*, so a cautious approach is mandatory. For example, a macro defined by **primarydef** might have disappeared by the time its second argument has been scanned; we solve this by increasing the reference count of its token list, so that the macro can be called even after it has been clobbered.

⟨ Declare the basic parsing subroutines 931 ⟩ +≡
  static void *mp_scan_secondary*(**MP** *mp*)
  {
    mp_node *p*;   /∗ for list manipulation ∗/
    halfword *c*, *d*;   /∗ operation codes or modifiers ∗/
    mp_node *cc* = Λ;
    mp_sym *mac_name* = Λ;   /∗ token defined with **primarydef** ∗/
  RESTART:
    if ((cur_cmd( ) < $mp\_min\_primary\_command$) ∨ (cur_cmd( ) > $mp\_max\_primary\_command$))
      mp_bad_exp(*mp*, "A␣secondary");
    ;
    mp_scan_primary(*mp*);
  CONTINUE:
    if (cur_cmd( ) ≤ $mp\_max\_secondary\_command$ ∧ cur_cmd( ) ≥ $mp\_min\_secondary\_command$) {
      p = *mp_stash_cur_exp*(*mp*);
      d = *cur_cmd*( );
      c = *cur_mod*( );
      if (*d* ≡ *mp_secondary_primary_macro*) {
        cc = *cur_mod_node*( );
        mac_name = *cur_sym*( );
        add_mac_ref(*cc*);
      }
      mp_get_x_next(*mp*);
      mp_scan_primary(*mp*);
      if (*d* ≠ *mp_secondary_primary_macro*) {
        mp_do_binary(*mp*, *p*, *c*);
      }
      else {
        mp_back_input(*mp*);
        mp_binary_mac(*mp*, *p*, *cc*, *mac_name*);
        decr_mac_ref(*cc*);
        mp_get_x_next(*mp*);
        goto RESTART;
      }
      goto CONTINUE;
    }
  }

945.   The following procedure calls a macro that has two parameters, p and cur_exp.

static void $mp\_binary\_mac(\textbf{MP}\ mp, \textbf{mp\_node}\ p, \textbf{mp\_node}\ c, \textbf{mp\_sym}\ n)$
{
   mp_node $q,\ r;$ /∗ nodes in the parameter list ∗/

   $q = mp\_get\_symbolic\_node(mp);$
   $r = mp\_get\_symbolic\_node(mp);$
   $mp\_link(q) = r;$
   $set\_mp\_sym\_sym(q, p);$
   $set\_mp\_sym\_sym(r, mp\_stash\_cur\_exp(mp));$
   $mp\_macro\_call(mp, c, q, n);$
}

946.   The next procedure, *scan_tertiary*, is pretty much the same deal.

⟨Declare the basic parsing subroutines 931⟩ +≡

  static void *mp_scan_tertiary*(**MP** *mp*)

  {

    mp_node *p*;   /∗ for list manipulation ∗/

    halfword *c*, *d*;   /∗ operation codes or modifiers ∗/

    mp_node *cc* = Λ;

    mp_sym *mac_name* = Λ;   /∗ token defined with **secondarydef** ∗/

  RESTART:

    if ((*cur_cmd*( ) < *mp_min_primary_command*) ∨ (*cur_cmd*( ) > *mp_max_primary_command*))

      mp_bad_exp(*mp*, "A␣tertiary");

    ;

    mp_scan_secondary(*mp*);

  CONTINUE:

    if (*cur_cmd*( ) ≤ *mp_max_tertiary_command*) {

      if (*cur_cmd*( ) ≥ *mp_min_tertiary_command*) {

        p = *mp_stash_cur_exp*(*mp*);

        c = *cur_mod*( );

        d = *cur_cmd*( );

        if (*d* ≡ *mp_tertiary_secondary_macro*) {

          cc = *cur_mod_node*( );

          mac_name = *cur_sym*( );

          add_mac_ref(*cc*);

        }

        mp_get_x_next(*mp*);

        mp_scan_secondary(*mp*);

        if (*d* ≠ *mp_tertiary_secondary_macro*) {

          mp_do_binary(*mp*, *p*, *c*);

        }

        else {

          mp_back_input(*mp*);

          mp_binary_mac(*mp*, *p*, *cc*, *mac_name*);

          decr_mac_ref(*cc*);

          mp_get_x_next(*mp*);

          goto RESTART;

        }

        goto CONTINUE;

      }

    }

  }

947.   Finally we reach the deepest level in our quartet of parsing routines. This one is much like the others; but it has an extra complication from paths, which materialize here.

⟨Declare the basic parsing subroutines 931⟩ +≡

```
static int mp_scan_path(MP mp);

static void mp_scan_expression(MP mp)
{
  int my_var_flag;       /* initial value of var_flag */

  my_var_flag = mp→var_flag;
  check_expansion_depth();
RESTART:
  if ((cur_cmd() < mp_min_primary_command) ∨ (cur_cmd() > mp_max_primary_command))
    mp_bad_exp(mp, "An");
  ;
  mp_scan_tertiary(mp);
CONTINUE:
  if (cur_cmd() ≤ mp_max_expression_command) {
    if (cur_cmd() ≥ mp_min_expression_command) {
      if ((cur_cmd() ≠ mp_equals) ∨ (my_var_flag ≠ mp_assignment)) {
        mp_node p;       /* for list manipulation */
        mp_node cc = Λ;
        halfword c;
        halfword d;      /* operation codes or modifiers */
        mp_sym mac_name;       /* token defined with tertiarydef */

        mac_name = Λ;
        p = mp_stash_cur_exp(mp);
        d = cur_cmd();
        c = cur_mod();
        if (d ≡ mp_expression_tertiary_macro) {
          cc = cur_mod_node();
          mac_name = cur_sym();
          add_mac_ref(cc);
        }
        if ((d < mp_ampersand) ∨ ((d ≡ mp_ampersand) ∧ ((mp_type(p) ≡ mp_pair_type) ∨ (mp_type(p) ≡
              mp_path_type)))) {
          /* Scan a path construction operation; but return if p has the wrong type */
          mp_unstash_cur_exp(mp, p);
          if (¬mp_scan_path(mp)) {
            mp→expand_depth_count −−;
            return;
          }
        }
      }
      else {
        mp_get_x_next(mp);
        mp_scan_tertiary(mp);
        if (d ≠ mp_expression_tertiary_macro) {
          mp_do_binary(mp, p, c);
        }
        else {
          mp_back_input(mp);
          mp_binary_mac(mp, p, cc, mac_name);
          decr_mac_ref(cc);
```

```
                    mp_get_x_next(mp);
                    goto RESTART;
                }
            }
            goto CONTINUE;
        }
    }
}
mp→expand_depth_count −−;
}
```

948.   The reader should review the data structure conventions for paths before hoping to understand the next part of this code.

#define $min\_tension$ $three\_quarter\_unit\_t$

⟨ Declare the basic parsing subroutines 931 ⟩ +≡

  static void $force\_valid\_tension\_setting($**MP** $mp)$

  {

    if $((mp\text{-}cur\_exp.type \neq mp\_known) \lor number\_less(cur\_exp\_value\_number(), min\_tension))$ {

      mp_value $new\_expr;$

      const char $*hlp[\,] = \{$"The␣expression␣above␣should␣have␣been␣a␣number␣>=3/4.", $\Lambda\};$

      $memset(\&new\_expr, 0, \textbf{sizeof}(\textbf{mp\_value}));$

      $new\_number(new\_expr.data.n);$

      $mp\_disp\_err(mp, \Lambda);$

      $number\_clone(new\_expr.data.n, unity\_t);$

      $mp\_back\_error(mp,$ "Improper␣tension␣has␣been␣set␣to␣1"$, hlp, true);$

      $mp\_get\_x\_next(mp);$

      $mp\_flush\_cur\_exp(mp, new\_expr);$

    }

  }

  static int $mp\_scan\_path($**MP** $mp)$

  {

    mp_knot $path\_p,$ $path\_q,$ $r;$

    mp_knot $pp,$ $qq;$

    halfword $d;$ /∗ operation code or modifier ∗/

    boolean $cycle\_hit;$ /∗ did a path expression just end with '**cycle**'? ∗/

    mp_number $x,$ $y;$ /∗ explicit coordinates or tension at a path join ∗/

    int $t;$ /∗ knot type following a path join ∗/

    $t = 0;$

    $cycle\_hit = false;$ /∗ Convert the left operand, p, into a partial path ending at q; but **return** if p doesn't have a suitable type ∗/

    if $(mp\text{-}cur\_exp.type \equiv mp\_pair\_type)$ $path\_p = mp\_pair\_to\_knot(mp);$

    else if $(mp\text{-}cur\_exp.type \equiv mp\_path\_type)$ $path\_p = cur\_exp\_knot();$

    else return 0;

    $path\_q = path\_p;$

    while $(mp\_next\_knot(path\_q) \neq path\_p)$ $path\_q = mp\_next\_knot(path\_q);$

    if $(mp\_left\_type(path\_p) \neq mp\_endpoint)$ {   /∗ open up a cycle ∗/

      $r = mp\_copy\_knot(mp, path\_p);$

      $mp\_next\_knot(path\_q) = r;$

      $path\_q = r;$

    }

    $mp\_left\_type(path\_p) = mp\_open;$

    $mp\_right\_type(path\_q) = mp\_open;$

    $new\_number(y);$

    $new\_number(x);$

  CONTINUE_PATH:

    /∗ Determine the path join parameters; but **goto** $finish\_path$ if there's only a direction specifier ∗/

    /∗ At this point cur_cmd is either $ampersand$, $left\_brace$, or $path\_join$. ∗/

    if $(cur\_cmd() \equiv mp\_left\_brace)$ {

      /∗ Put the pre-join direction information into node q ∗/   /∗ At this point $mp\_right\_type(q)$ is usually $open$, but it may have been set to some other value by a previous operation. We must maintain the value of $mp\_right\_type(q)$ in cases such as '..{curl2}z{0,0}..'. ∗/

      $t = mp\_scan\_direction(mp);$

```
  if (t ≠ mp_open) {
    mp_right_type(path_q) = (unsigned short) t;
    number_clone(path_q→right_given, cur_exp_value_number( ));
    if (mp_left_type(path_q) ≡ mp_open) {
      mp_left_type(path_q) = (unsigned short) t;
      number_clone(path_q→left_given, cur_exp_value_number( ));
    }    /* note that left_given(q) = left_curl(q) */
  }
}
d = cur_cmd( );
if (d ≡ mp_path_join) {        /* Determine the tension and/or control points */
  mp_get_x_next(mp);
  if (cur_cmd( ) ≡ mp_tension) {      /* Set explicit tensions */
    mp_get_x_next(mp);
    set_number_from_scaled(y, cur_cmd( ));
    if (cur_cmd( ) ≡ mp_at_least)  mp_get_x_next(mp);
    mp_scan_primary(mp);
    force_valid_tension_setting(mp);
    if (number_to_scaled(y) ≡ mp_at_least) {
      if (is_number(cur_exp_value_number( )))  number_negate(cur_exp_value_number( ));
    }
    number_clone(path_q→right_tension, cur_exp_value_number( ));
    if (cur_cmd( ) ≡ mp_and_command) {
      mp_get_x_next(mp);
      set_number_from_scaled(y, cur_cmd( ));
      if (cur_cmd( ) ≡ mp_at_least)  mp_get_x_next(mp);
      mp_scan_primary(mp);
      force_valid_tension_setting(mp);
      if (number_to_scaled(y) ≡ mp_at_least) {
        if (is_number(cur_exp_value_number( )))  number_negate(cur_exp_value_number( ));
      }
    }
    number_clone(y, cur_exp_value_number( ));
  }
  else if (cur_cmd( ) ≡ mp_controls) {      /* Set explicit control points */
    mp_right_type(path_q) = mp_explicit;
    t = mp_explicit;
    mp_get_x_next(mp);
    mp_scan_primary(mp);
    mp_known_pair(mp);
    number_clone(path_q→right_x, mp→cur_x);
    number_clone(path_q→right_y, mp→cur_y);
    if (cur_cmd( ) ≠ mp_and_command) {
      number_clone(x, path_q→right_x);
      number_clone(y, path_q→right_y);
    }
    else {
      mp_get_x_next(mp);
      mp_scan_primary(mp);
      mp_known_pair(mp);
      number_clone(x, mp→cur_x);
      number_clone(y, mp→cur_y);
```

```
        }
      }
      else {
        set_number_to_unity (path_q→right_tension );
        set_number_to_unity (y);
        mp_back_input (mp);        /∗ default tension ∗/
        goto DONE;
      }
      ;
      if (cur_cmd ( ) ≠ mp_path_join ) {
        const char ∗hlp[ ] = {"A␣path␣join␣command␣should␣end␣with␣two␣dots.", Λ};
        mp_back_error (mp, "Missing␣`..'␣has␣been␣inserted", hlp, true );
      }
    DONE: ;
    }
    else if (d ≠ mp_ampersand ) {
      goto FINISH_PATH;
    }
    mp_get_x_next (mp);
    if (cur_cmd ( ) ≡ mp_left_brace ) {        /∗ Put the post-join direction information into x and t ∗/
        /∗ Since left_tension and mp_left_y share the same position in knot nodes, and since left_given is
            similarly equivalent to left_x, we use x and y to hold the given direction and tension information
            when there are no explicit control points. ∗/
      t = mp_scan_direction (mp);
      if (mp_right_type (path_q ) ≠ mp_explicit ) number_clone (x, cur_exp_value_number ( ));
      else  t = mp_explicit ;        /∗ the direction information is superfluous ∗/
    }
    else if (mp_right_type (path_q ) ≠ mp_explicit ) {
      t = mp_open ;
      set_number_to_zero (x);
    }
    if (cur_cmd ( ) ≡ mp_cycle ) {        /∗ Get ready to close a cycle ∗/
        /∗ If a person tries to define an entire path by saying '(x,y)&cycle', we silently change the
            specification to '(x,y)..cycle', since a cycle shouldn't have length zero. ∗/
      cycle_hit = true ;
      mp_get_x_next (mp);
      pp = path_p ;
      qq = path_p ;
      if (d ≡ mp_ampersand ) {
        if (path_p ≡ path_q ) {
          d = mp_path_join ;
          set_number_to_unity (path_q→right_tension );
          set_number_to_unity (y);
        }
      }
    }
    else {
      mp_scan_tertiary (mp);
        /∗ Convert the right operand, cur_exp, into a partial path from pp to qq ∗/
      if (mp→cur_exp.type ≠ mp_path_type ) pp = mp_pair_to_knot (mp);
      else  pp = cur_exp_knot ( );
      qq = pp ;
```

```
      while (mp_next_knot(qq) ≠ pp)  qq = mp_next_knot(qq);
      if (mp_left_type(pp) ≠ mp_endpoint) {        /* open up a cycle */
        r = mp_copy_knot(mp, pp);
        mp_next_knot(qq) = r;
        qq = r;
      }
      mp_left_type(pp) = mp_open;
      mp_right_type(qq) = mp_open;
    }     /* Join the partial paths and reset p and q to the head and tail of the result */
    if (d ≡ mp_ampersand) {
      if (¬(number_equal(path_q→x_coord, pp→x_coord)) ∨ ¬(number_equal(path_q→y_coord, pp→y_coord))) {
        const char *hlp[] = {"When␣you␣join␣paths␣'p&q',␣the␣ending␣point␣of␣p",
            "must␣be␣exactly␣equal␣to␣the␣starting␣point␣of␣q.",
            "So␣I'm␣going␣to␣pretend␣that␣you␣said␣'p..q'␣instead.", Λ};

        mp_back_error(mp, "Paths␣don't␣touch;␣'&'␣will␣be␣changed␣to␣'..'", hlp, true);
        ;
        mp_get_x_next(mp);
        d = mp_path_join;
        set_number_to_unity(path_q→right_tension);
        set_number_to_unity(y);
      }
    }     /* Plug an opening in mp_right_type(pp), if possible */
    if (mp_right_type(pp) ≡ mp_open) {
      if ((t ≡ mp_curl) ∨ (t ≡ mp_given)) {
        mp_right_type(pp) = (unsigned short) t;
        number_clone(pp→right_given, x);
      }
    }
    if (d ≡ mp_ampersand) {        /* Splice independent paths together */
      if (mp_left_type(path_q) ≡ mp_open)
        if (mp_right_type(path_q) ≡ mp_open) {
          mp_left_type(path_q) = mp_curl;
          set_number_to_unity(path_q→left_curl);
        }
      if (mp_right_type(pp) ≡ mp_open)
        if (t ≡ mp_open) {
          mp_right_type(pp) = mp_curl;
          set_number_to_unity(pp→right_curl);
        }
      mp_right_type(path_q) = mp_right_type(pp);
      mp_next_knot(path_q) = mp_next_knot(pp);
      number_clone(path_q→right_x, pp→right_x);
      number_clone(path_q→right_y, pp→right_y);
      mp_xfree(pp);
      if (qq ≡ pp)  qq = path_q;
    }
    else {        /* Plug an opening in mp_right_type(q), if possible */
      if (mp_right_type(path_q) ≡ mp_open) {
        if ((mp_left_type(path_q) ≡ mp_curl) ∨ (mp_left_type(path_q) ≡ mp_given)) {
          mp_right_type(path_q) = mp_left_type(path_q);
          number_clone(path_q→right_given, path_q→left_given);
        }
```

```
      }
      mp_next_knot(path_q) = pp;
      number_clone(pp→left_y, y);
      if (t ≠ mp_open) {
        number_clone(pp→left_x, x);
        mp_left_type(pp) = (unsigned short) t;
      }
      ;
    }
    path_q = qq;
    if (cur_cmd( ) ≥ mp_min_expression_command)
      if (cur_cmd( ) ≤ mp_ampersand)
        if (¬cycle_hit) goto CONTINUE_PATH;
  FINISH_PATH:    /∗ Choose control points for the path and put the result into cur_exp ∗/
    if (cycle_hit) {
      if (d ≡ mp_ampersand) path_p = path_q;
    }
    else {
      mp_left_type(path_p) = mp_endpoint;
      if (mp_right_type(path_p) ≡ mp_open) {
        mp_right_type(path_p) = mp_curl;
        set_number_to_unity(path_p→right_curl);
      }
      mp_right_type(path_q) = mp_endpoint;
      if (mp_left_type(path_q) ≡ mp_open) {
        mp_left_type(path_q) = mp_curl;
        set_number_to_unity(path_q→left_curl);
      }
      mp_next_knot(path_q) = path_p;
    }
    mp_make_choices(mp, path_p);
    mp→cur_exp.type = mp_path_type;
    set_cur_exp_knot(path_p);
    free_number(x);
    free_number(y);
    return 1;
  }
```

949.   A pair of numeric values is changed into a knot node for a one-point path when METAPOST discovers
that the pair is part of a path.

> **static mp_knot** *mp_pair_to_knot*(**MP** *mp*)
> {   /* convert a pair to a knot with two endpoints */
>   **mp_knot** *q*;   /* the new node */
>
>   *q* = *mp_new_knot*(*mp*);
>   *mp_left_type*(*q*) = *mp_endpoint*;
>   *mp_right_type*(*q*) = *mp_endpoint*;
>   *mp_originator*(*q*) = *mp_metapost_user*;
>   *mp_next_knot*(*q*) = *q*;
>   *mp_known_pair*(*mp*);
>   *number_clone*(*q*→*x_coord*, *mp*→*cur_x*);
>   *number_clone*(*q*→*y_coord*, *mp*→*cur_y*);
>   **return** *q*;
> }

950.   The *known_pair* subroutine sets *cur_x* and *cur_y* to the components of the current expression,
assuming that the current expression is a pair of known numerics. Unknown components are zeroed, and
the current expression is flushed.

⟨ Declarations 8 ⟩ +≡
  static void *mp_known_pair*(**MP** *mp*);

951.   **void** *mp_known_pair* (**MP** *mp*)
{
    mp_value *new_expr*;
    mp_node *p*;   /∗ the pair node ∗/

    memset (&*new_expr*, 0, **sizeof** (**mp_value**));
    new_number (*new_expr*.*data*.*n*);
    if (*mp*→*cur_exp*.*type* ≠ *mp_pair_type*) {
        const char ∗*hlp*[ ] = {"I␣need␣x␣and␣y␣numbers␣for␣this␣part␣of␣the␣path.",
            "The␣value␣I␣found␣(see␣above)␣was␣no␣good;",
            "so␣I'll␣try␣to␣keep␣going␣by␣using␣zero␣instead.",
            "(Chapter␣27␣of␣The␣METAFONTbook␣explains␣that",
            "you␣might␣want␣to␣type␣'I␣??""?'␣now.)", Λ};
      mp_disp_err (*mp*, Λ);
      mp_back_error (*mp*, "Undefined␣coordinates␣have␣been␣replaced␣by␣(0,0)", *hlp*, *true*);
      mp_get_x_next (*mp*);
      mp_flush_cur_exp (*mp*, *new_expr*);
      set_number_to_zero (*mp*→*cur_x*);
      set_number_to_zero (*mp*→*cur_y*);
    }
    else {
      p = *value_node* (*cur_exp_node* ( ));
        /∗ Make sure that both *x* and *y* parts of *p* are known; copy them into *cur_x* and *cur_y* ∗/
      if (*mp_type* (*x_part* (*p*)) ≡ *mp_known*) {
        number_clone (*mp*→*cur_x*, *value_number* (*x_part* (*p*)));
      }
      else {
        const char ∗*hlp*[ ] = {"I␣need␣a␣'known'␣x␣value␣for␣this␣part␣of␣the␣path.",
            "The␣value␣I␣found␣(see␣above)␣was␣no␣good;",
            "so␣I'll␣try␣to␣keep␣going␣by␣using␣zero␣instead.",
            "(Chapter␣27␣of␣The␣METAFONTbook␣explains␣that",
            "you␣might␣want␣to␣type␣'I␣??""?'␣now.)", Λ};
        mp_disp_err (*mp*, *x_part* (*p*));
        mp_back_error (*mp*, "Undefined␣x␣coordinate␣has␣been␣replaced␣by␣0", *hlp*, *true*);
        mp_get_x_next (*mp*);
        mp_recycle_value (*mp*, *x_part* (*p*));
        set_number_to_zero (*mp*→*cur_x*);
      }
      if (*mp_type* (*y_part* (*p*)) ≡ *mp_known*) {
        number_clone (*mp*→*cur_y*, *value_number* (*y_part* (*p*)));
      }
      else {
        const char ∗*hlp*[ ] = {"I␣need␣a␣'known'␣y␣value␣for␣this␣part␣of␣the␣path.",
            "The␣value␣I␣found␣(see␣above)␣was␣no␣good;",
            "so␣I'll␣try␣to␣keep␣going␣by␣using␣zero␣instead.",
            "(Chapter␣27␣of␣The␣METAFONTbook␣explains␣that",
            "you␣might␣want␣to␣type␣'I␣??""?'␣now.)", Λ};
        mp_disp_err (*mp*, *y_part* (*p*));
        mp_back_error (*mp*, "Undefined␣y␣coordinate␣has␣been␣replaced␣by␣0", *hlp*, *true*);
        mp_get_x_next (*mp*);
        mp_recycle_value (*mp*, *y_part* (*p*));
        set_number_to_zero (*mp*→*cur_y*);

```
    }
    mp_flush_cur_exp(mp, new_expr);
  }
}
```

952.   The *scan_direction* subroutine looks at the directional information that is enclosed in braces, and also scans ahead to the following character. A type code is returned, either *open* (if the direction was $(0,0)$), or *curl* (if the direction was a curl of known value *cur_exp*), or *given* (if the direction is given by the *angle* value that now appears in *cur_exp*).

There's nothing difficult about this subroutine, but the program is rather lengthy because a variety of potential errors need to be nipped in the bud.

static quarterword *mp_scan_direction*(**MP** *mp*)
{
  int *t*;   /∗ the type of information found ∗/
  mp_get_x_next(*mp*);
  if (*cur_cmd*( ) ≡ *mp_curl_command*) {   /∗ Scan a curl specification ∗/
    mp_get_x_next(*mp*);
    mp_scan_expression(*mp*);
    if ((*mp*→*cur_exp.type* ≠ *mp_known*) ∨ (*number_negative*(*cur_exp_value_number*( )))) {
      mp_value *new_expr*;
      const char ∗*hlp*[ ] = {"A␣curl␣must␣be␣a␣known,␣nonnegative␣number.", Λ};

      memset(&*new_expr*, 0, **sizeof**(**mp_value**));
      new_number(*new_expr.data.n*);
      set_number_to_unity(*new_expr.data.n*);
      mp_disp_err(*mp*, Λ);
      mp_back_error(*mp*, "Improper␣curl␣has␣been␣replaced␣by␣1", *hlp*, *true*);
      mp_get_x_next(*mp*);
      mp_flush_cur_exp(*mp*, *new_expr*);
    }
    t = *mp_curl*;
  }
  else {   /∗ Scan a given direction ∗/
    mp_scan_expression(*mp*);
    if (*mp*→*cur_exp.type* > *mp_pair_type*) {   /∗ Get given directions separated by commas ∗/
      mp_number *xx*;

      new_number(*xx*);
      if (*mp*→*cur_exp.type* ≠ *mp_known*) {
        mp_value *new_expr*;
        const char ∗*hlp*[ ] = {"I␣need␣a␣'known'␣x␣value␣for␣this␣part␣of␣the␣path.",
            "The␣value␣I␣found␣(see␣above)␣was␣no␣good;",
            "so␣I'll␣try␣to␣keep␣going␣by␣using␣zero␣instead.",
            "(Chapter␣27␣of␣The␣METAFONTbook␣explains␣that",
            "you␣might␣want␣to␣type␣'I␣??""?'␣now.)", Λ};
        memset(&*new_expr*, 0, **sizeof**(**mp_value**));
        new_number(*new_expr.data.n*);
        set_number_to_zero(*new_expr.data.n*);
        mp_disp_err(*mp*, Λ);
        mp_back_error(*mp*, "Undefined␣x␣coordinate␣has␣been␣replaced␣by␣0", *hlp*, *true*);
        mp_get_x_next(*mp*);
        mp_flush_cur_exp(*mp*, *new_expr*);
      }
      number_clone(*xx*, *cur_exp_value_number*( ));
      if (*cur_cmd*( ) ≠ *mp_comma*) {
        const char ∗*hlp*[ ] = {"I've␣got␣the␣x␣coordinate␣of␣a␣path␣direction;",
            "will␣look␣for␣the␣y␣coordinate␣next.", Λ};
        mp_back_error(*mp*, "Missing␣','␣has␣been␣inserted", *hlp*, *true*);

```
      }
    mp_get_x_next(mp);
    mp_scan_expression(mp);
    if (mp→cur_exp.type ≠ mp_known) {
      mp_value new_expr;
      const char *hlp[] = {"I␣need␣a␣'known'␣y␣value␣for␣this␣part␣of␣the␣path.",
          "The␣value␣I␣found␣(see␣above)␣was␣no␣good;",
          "so␣I'll␣try␣to␣keep␣going␣by␣using␣zero␣instead.",
          "(Chapter␣27␣of␣The␣METAFONTbook␣explains␣that",
          "you␣might␣want␣to␣type␣'I␣??""?'␣now.)", Λ};
      memset(&new_expr, 0, sizeof(mp_value));
      new_number(new_expr.data.n);
      set_number_to_zero(new_expr.data.n);
      mp_disp_err(mp, Λ);
      mp_back_error(mp, "Undefined␣y␣coordinate␣has␣been␣replaced␣by␣0", hlp, true);
      mp_get_x_next(mp);
      mp_flush_cur_exp(mp, new_expr);
    }
    number_clone(mp→cur_y, cur_exp_value_number());
    number_clone(mp→cur_x, xx);
    free_number(xx);
  }
  else {
    mp_known_pair(mp);
  }
  if (number_zero(mp→cur_x) ∧ number_zero(mp→cur_y)) t = mp_open;
  else {
    mp_number narg;
    new_angle(narg);
    n_arg(narg, mp→cur_x, mp→cur_y);
    t = mp_given;
    set_cur_exp_value_number(narg);
    free_number(narg);
  }
}
if (cur_cmd() ≠ mp_right_brace) {
  const char *hlp[] = {"I've␣scanned␣a␣direction␣spec␣for␣part␣of␣a␣path,",
      "so␣a␣right␣brace␣should␣have␣come␣next.", "I␣shall␣pretend␣that␣one␣was␣there.",
      Λ};
  mp_back_error(mp, "Missing␣'}'␣has␣been␣inserted", hlp, true);
}
mp_get_x_next(mp);
return (quarterword) t;
}
```

953.   Finally, we sometimes need to scan an expression whose value is supposed to be either *true_code* or *false_code*.

#define *mp_get_boolean*(*mp*)   **do**
```
      {
         mp_get_x_next(mp);
         mp_scan_expression(mp);
         if (mp→cur_exp.type ≠ mp_boolean_type) {
            do_boolean_error(mp);
         }
      }
      while (0)
```
⟨ Declare the basic parsing subroutines 931 ⟩ +≡
```
  static void do_boolean_error(MP mp)
  {
    mp_value new_expr;
    const char *hlp[] = {"The␣expression␣shown␣above␣should␣have␣had␣a␣definite",
        "true-or-false␣value.␣I'm␣changing␣it␣to␣'false'.", Λ};

    memset(&new_expr, 0, sizeof(mp_value));
    new_number(new_expr.data.n);
    mp_disp_err(mp, Λ);
    set_number_from_boolean(new_expr.data.n, mp_false_code);
    mp_back_error(mp, "Undefined␣condition␣will␣be␣treated␣as␣'false'", hlp, true);
    mp_get_x_next(mp);
    mp_flush_cur_exp(mp, new_expr);
    mp→cur_exp.type = mp_boolean_type;
  }
```

954.   ⟨ Declarations 8 ⟩ +≡
```
  static void do_boolean_error(MP mp);
```

955.   Doing the operations.   The purpose of parsing is primarily to permit people to avoid piles of parentheses. But the real work is done after the structure of an expression has been recognized; that's when new expressions are generated. We turn now to the guts of METAPOST, which handles individual operators that have come through the parsing mechanism.

We'll start with the easy ones that take no operands, then work our way up to operators with one and ultimately two arguments. In other words, we will write the three procedures *do_nullary*, *do_unary*, and *do_binary* that are invoked periodically by the expression scanners.

First let's make sure that all of the primitive operators are in the hash table. Although *scan_primary* and its relatives made use of the *cmd* code for these operators, the *do* routines base everything on the *mod* code. For example, *do_binary* doesn't care whether the operation it performs is a *primary_binary* or *secondary_binary*, etc.

⟨ Put each of METAPOST's primitives into the hash table 200 ⟩ +≡

  mp_primitive(*mp*, "true", *mp_nullary*, *mp_true_code*);
  ;
  mp_primitive(*mp*, "false", *mp_nullary*, *mp_false_code*);
  ;
  mp_primitive(*mp*, "nullpicture", *mp_nullary*, *mp_null_picture_code*);
  ;
  mp_primitive(*mp*, "nullpen", *mp_nullary*, *mp_null_pen_code*);
  ;
  mp_primitive(*mp*, "readstring", *mp_nullary*, *mp_read_string_op*);
  ;
  mp_primitive(*mp*, "pencircle", *mp_nullary*, *mp_pen_circle*);
  ;
  mp_primitive(*mp*, "normaldeviate", *mp_nullary*, *mp_normal_deviate*);
  ;
  mp_primitive(*mp*, "readfrom", *mp_unary*, *mp_read_from_op*);
  ;
  mp_primitive(*mp*, "closefrom", *mp_unary*, *mp_close_from_op*);
  ;
  mp_primitive(*mp*, "odd", *mp_unary*, *mp_odd_op*);
  ;
  mp_primitive(*mp*, "known", *mp_unary*, *mp_known_op*);
  ;
  mp_primitive(*mp*, "unknown", *mp_unary*, *mp_unknown_op*);
  ;
  mp_primitive(*mp*, "not", *mp_unary*, *mp_not_op*);
  ;
  mp_primitive(*mp*, "decimal", *mp_unary*, *mp_decimal*);
  ;
  mp_primitive(*mp*, "reverse", *mp_unary*, *mp_reverse*);
  ;
  mp_primitive(*mp*, "makepath", *mp_unary*, *mp_make_path_op*);
  ;
  mp_primitive(*mp*, "makepen", *mp_unary*, *mp_make_pen_op*);
  ;
  mp_primitive(*mp*, "oct", *mp_unary*, *mp_oct_op*);
  ;
  mp_primitive(*mp*, "hex", *mp_unary*, *mp_hex_op*);
  ;
  mp_primitive(*mp*, "ASCII", *mp_unary*, *mp_ASCII_op*);
  ;

$mp\_primitive(mp, \texttt{"char"}, mp\_unary, mp\_char\_op);$
;
$mp\_primitive(mp, \texttt{"length"}, mp\_unary, mp\_length\_op);$
;
$mp\_primitive(mp, \texttt{"turningnumber"}, mp\_unary, mp\_turning\_op);$
;
$mp\_primitive(mp, \texttt{"xpart"}, mp\_unary, mp\_x\_part);$
;
$mp\_primitive(mp, \texttt{"ypart"}, mp\_unary, mp\_y\_part);$
;
$mp\_primitive(mp, \texttt{"xxpart"}, mp\_unary, mp\_xx\_part);$
;
$mp\_primitive(mp, \texttt{"xypart"}, mp\_unary, mp\_xy\_part);$
;
$mp\_primitive(mp, \texttt{"yxpart"}, mp\_unary, mp\_yx\_part);$
;
$mp\_primitive(mp, \texttt{"yypart"}, mp\_unary, mp\_yy\_part);$
;
$mp\_primitive(mp, \texttt{"redpart"}, mp\_unary, mp\_red\_part);$
;
$mp\_primitive(mp, \texttt{"greenpart"}, mp\_unary, mp\_green\_part);$
;
$mp\_primitive(mp, \texttt{"bluepart"}, mp\_unary, mp\_blue\_part);$
;
$mp\_primitive(mp, \texttt{"cyanpart"}, mp\_unary, mp\_cyan\_part);$
;
$mp\_primitive(mp, \texttt{"magentapart"}, mp\_unary, mp\_magenta\_part);$
;
$mp\_primitive(mp, \texttt{"yellowpart"}, mp\_unary, mp\_yellow\_part);$
;
$mp\_primitive(mp, \texttt{"blackpart"}, mp\_unary, mp\_black\_part);$
;
$mp\_primitive(mp, \texttt{"greypart"}, mp\_unary, mp\_grey\_part);$
;
$mp\_primitive(mp, \texttt{"colormodel"}, mp\_unary, mp\_color\_model\_part);$
;
$mp\_primitive(mp, \texttt{"fontpart"}, mp\_unary, mp\_font\_part);$
;
$mp\_primitive(mp, \texttt{"textpart"}, mp\_unary, mp\_text\_part);$
;
$mp\_primitive(mp, \texttt{"prescriptpart"}, mp\_unary, mp\_prescript\_part);$
;
$mp\_primitive(mp, \texttt{"postscriptpart"}, mp\_unary, mp\_postscript\_part);$
;
$mp\_primitive(mp, \texttt{"pathpart"}, mp\_unary, mp\_path\_part);$
;
$mp\_primitive(mp, \texttt{"penpart"}, mp\_unary, mp\_pen\_part);$
;
$mp\_primitive(mp, \texttt{"dashpart"}, mp\_unary, mp\_dash\_part);$
;
$mp\_primitive(mp, \texttt{"sqrt"}, mp\_unary, mp\_sqrt\_op);$
;

$mp\_primitive(mp, \texttt{"mexp"}, mp\_unary, mp\_m\_exp\_op);$
;
$mp\_primitive(mp, \texttt{"mlog"}, mp\_unary, mp\_m\_log\_op);$
;
$mp\_primitive(mp, \texttt{"sind"}, mp\_unary, mp\_sin\_d\_op);$
;
$mp\_primitive(mp, \texttt{"cosd"}, mp\_unary, mp\_cos\_d\_op);$
;
$mp\_primitive(mp, \texttt{"floor"}, mp\_unary, mp\_floor\_op);$
;
$mp\_primitive(mp, \texttt{"uniformdeviate"}, mp\_unary, mp\_uniform\_deviate);$
;
$mp\_primitive(mp, \texttt{"charexists"}, mp\_unary, mp\_char\_exists\_op);$
;
$mp\_primitive(mp, \texttt{"fontsize"}, mp\_unary, mp\_font\_size);$
;
$mp\_primitive(mp, \texttt{"llcorner"}, mp\_unary, mp\_ll\_corner\_op);$
;
$mp\_primitive(mp, \texttt{"lrcorner"}, mp\_unary, mp\_lr\_corner\_op);$
;
$mp\_primitive(mp, \texttt{"ulcorner"}, mp\_unary, mp\_ul\_corner\_op);$
;
$mp\_primitive(mp, \texttt{"urcorner"}, mp\_unary, mp\_ur\_corner\_op);$
;
$mp\_primitive(mp, \texttt{"arclength"}, mp\_unary, mp\_arc\_length);$
;
$mp\_primitive(mp, \texttt{"angle"}, mp\_unary, mp\_angle\_op);$
;
$mp\_primitive(mp, \texttt{"cycle"}, mp\_cycle, mp\_cycle\_op);$
;
$mp\_primitive(mp, \texttt{"stroked"}, mp\_unary, mp\_stroked\_op);$
;
$mp\_primitive(mp, \texttt{"filled"}, mp\_unary, mp\_filled\_op);$
;
$mp\_primitive(mp, \texttt{"textual"}, mp\_unary, mp\_textual\_op);$
;
$mp\_primitive(mp, \texttt{"clipped"}, mp\_unary, mp\_clipped\_op);$
;
$mp\_primitive(mp, \texttt{"bounded"}, mp\_unary, mp\_bounded\_op);$
;
$mp\_primitive(mp, \texttt{"+"}, mp\_plus\_or\_minus, mp\_plus);$
;
$mp\_primitive(mp, \texttt{"-"}, mp\_plus\_or\_minus, mp\_minus);$
;
$mp\_primitive(mp, \texttt{"*"}, mp\_secondary\_binary, mp\_times);$
;
$mp\_primitive(mp, \texttt{"/"}, mp\_slash, mp\_over);$
$mp\text{-}frozen\_slash = mp\_frozen\_primitive(mp, \texttt{"/"}, mp\_slash, mp\_over);$
;
$mp\_primitive(mp, \texttt{"++"}, mp\_tertiary\_binary, mp\_pythag\_add);$
;
$mp\_primitive(mp, \texttt{"+-+"}, mp\_tertiary\_binary, mp\_pythag\_sub);$

;
$mp\_primitive\,(mp, \texttt{"or"}, mp\_tertiary\_binary, mp\_or\_op\,);$
;
$mp\_primitive\,(mp, \texttt{"and"}, mp\_and\_command, mp\_and\_op\,);$
;
$mp\_primitive\,(mp, \texttt{"<"}, mp\_expression\_binary, mp\_less\_than\,);$
;
$mp\_primitive\,(mp, \texttt{"<="}, mp\_expression\_binary, mp\_less\_or\_equal\,);$
;
$mp\_primitive\,(mp, \texttt{">"}, mp\_expression\_binary, mp\_greater\_than\,);$
;
$mp\_primitive\,(mp, \texttt{">="}, mp\_expression\_binary, mp\_greater\_or\_equal\,);$
;
$mp\_primitive\,(mp, \texttt{"="}, mp\_equals, mp\_equal\_to\,);$
;
$mp\_primitive\,(mp, \texttt{"<>"}, mp\_expression\_binary, mp\_unequal\_to\,);$
;
$mp\_primitive\,(mp, \texttt{"substring"}, mp\_primary\_binary, mp\_substring\_of\,);$
;
$mp\_primitive\,(mp, \texttt{"subpath"}, mp\_primary\_binary, mp\_subpath\_of\,);$
;
$mp\_primitive\,(mp, \texttt{"directiontime"}, mp\_primary\_binary, mp\_direction\_time\_of\,);$
;
$mp\_primitive\,(mp, \texttt{"point"}, mp\_primary\_binary, mp\_point\_of\,);$
;
$mp\_primitive\,(mp, \texttt{"precontrol"}, mp\_primary\_binary, mp\_precontrol\_of\,);$
;
$mp\_primitive\,(mp, \texttt{"postcontrol"}, mp\_primary\_binary, mp\_postcontrol\_of\,);$
;
$mp\_primitive\,(mp, \texttt{"penoffset"}, mp\_primary\_binary, mp\_pen\_offset\_of\,);$
;
$mp\_primitive\,(mp, \texttt{"arctime"}, mp\_primary\_binary, mp\_arc\_time\_of\,);$
;
$mp\_primitive\,(mp, \texttt{"mpversion"}, mp\_nullary, mp\_version\,);$
;
$mp\_primitive\,(mp, \texttt{"&"}, mp\_ampersand, mp\_concatenate\,);$
;
$mp\_primitive\,(mp, \texttt{"rotated"}, mp\_secondary\_binary, mp\_rotated\_by\,);$
;
$mp\_primitive\,(mp, \texttt{"slanted"}, mp\_secondary\_binary, mp\_slanted\_by\,);$
;
$mp\_primitive\,(mp, \texttt{"scaled"}, mp\_secondary\_binary, mp\_scaled\_by\,);$
;
$mp\_primitive\,(mp, \texttt{"shifted"}, mp\_secondary\_binary, mp\_shifted\_by\,);$
;
$mp\_primitive\,(mp, \texttt{"transformed"}, mp\_secondary\_binary, mp\_transformed\_by\,);$
;
$mp\_primitive\,(mp, \texttt{"xscaled"}, mp\_secondary\_binary, mp\_x\_scaled\,);$
;
$mp\_primitive\,(mp, \texttt{"yscaled"}, mp\_secondary\_binary, mp\_y\_scaled\,);$
;
$mp\_primitive\,(mp, \texttt{"zscaled"}, mp\_secondary\_binary, mp\_z\_scaled\,);$

;
$mp\_primitive(mp, "infont", mp\_secondary\_binary, mp\_in\_font);$
;
$mp\_primitive(mp, "intersectiontimes", mp\_tertiary\_binary, mp\_intersect);$
;
$mp\_primitive(mp, "envelope", mp\_primary\_binary, mp\_envelope\_of);$
;
$mp\_primitive(mp, "glyph", mp\_primary\_binary, mp\_glyph\_infont);$

956.   ⟨ Cases of *print_cmd_mod* for symbolic printing of primitives 233 ⟩ +≡
case $mp\_nullary:$ **case** $mp\_unary:$ **case** $mp\_primary\_binary:$ **case** $mp\_secondary\_binary:$
   case $mp\_tertiary\_binary:$ **case** $mp\_expression\_binary:$ **case** $mp\_cycle:$ **case** $mp\_plus\_or\_minus:$
   case $mp\_slash:$ **case** $mp\_ampersand:$ **case** $mp\_equals:$ **case** $mp\_and\_command:$
   $mp\_print\_op(mp, (\mathbf{quarterword})\ m);$
   break;

957.   OK, let's look at the simplest *do* procedure first.

⟨ Declare nullary action procedure 958 ⟩;

```
static void mp_do_nullary(MP mp, quarterword c)
{
  check_arith();
  if (number_greater(internal_value(mp_tracing_commands), two_t))
    mp_show_cmd_mod(mp, mp_nullary, c);
  switch (c) {
  case mp_true_code: case mp_false_code: mp→cur_exp.type = mp_boolean_type;
    set_cur_exp_value_boolean(c);
    break;
  case mp_null_picture_code: mp→cur_exp.type = mp_picture_type;
    set_cur_exp_node((mp_node) mp_get_edge_header_node(mp));
    mp_init_edges(mp, (mp_edge_header_node) cur_exp_node());
    break;
  case mp_null_pen_code: mp→cur_exp.type = mp_pen_type;
    set_cur_exp_knot(mp_get_pen_circle(mp, zero_t));
    break;
  case mp_normal_deviate:
    {
      mp_number r;

      new_number(r);
      mp_norm_rand(mp, &r);
      mp→cur_exp.type = mp_known;
      set_cur_exp_value_number(r);
      free_number(r);
    }
    break;
  case mp_pen_circle: mp→cur_exp.type = mp_pen_type;
    set_cur_exp_knot(mp_get_pen_circle(mp, unity_t));
    break;
  case mp_version: mp→cur_exp.type = mp_string_type;
    set_cur_exp_str(mp_intern(mp, metapost_version));
    break;
  case mp_read_string_op:      /* Read a string from the terminal */
    if (mp→noninteractive ∨ mp→interaction ≤ mp_nonstop_mode)
      mp_fatal_error(mp, "***␣(cannot␣readstring␣in␣nonstop␣modes)");
    mp_begin_file_reading(mp);
    name = is_read;
    limit = start;
    prompt_input("");
    mp_finish_read(mp);
    break;
  }      /* there are no other cases */
  check_arith();
}
```

958.   ⟨ Declare nullary action procedure 958 ⟩ ≡

  static void *mp_finish_read* (**MP** *mp*)

  {   /∗ copy *buffer* line to *cur_exp* ∗/

    size_t *k*;

    str_room(((**int**) *mp*→*last* − (**int**) *start*));

    for (*k* = (**size_t**) *start*; *k* < *mp*→*last*; *k*++) {

      append_char (*mp*→*buffer* [*k*]);

    }

    mp_end_file_reading (*mp*);

    mp→*cur_exp.type* = *mp_string_type*;

    set_cur_exp_str (*mp_make_string* (*mp*));

  }

This code is used in section 957.

959.   Things get a bit more interesting when there's an operand. The operand to *do_unary* appears in *cur_type* and *cur_exp*.

This complicated if test makes sure that any *bounds* or *clip* picture objects that get passed into **within** do not raise an error when queried using the color part primitives (this is needed for backward compatibility) .

#define *cur_pic_item*   *mp_link*(*edge_list*(*cur_exp_node*()))
#define *pict_color_type*(*A*)
      ((*cur_pic_item* ≠ Λ) ∧ ((¬*has_color*(*cur_pic_item*)) ∨ (((**mp_color_model**(*cur_pic_item*) ≡
          A) ∨ ((**mp_color_model**(*cur_pic_item*) ≡ *mp_uninitialized_model*) ∧
          (*number_to_scaled*(*internal_value*(*mp_default_color_model*))/*number_to_scaled*(*unity_t*)) ≡
          (*A*))))))
#define *boolean_reset*(*A*)
      if ((*A*)) *set_cur_exp_value_boolean*(*mp_true_code*);
      else *set_cur_exp_value_boolean*(*mp_false_code*)
#define *type_range*(*A*, *B*)
      {
        if ((*mp*→*cur_exp*.*type* ≥ (*A*)) ∧ (*mp*→*cur_exp*.*type* ≤ (*B*)))
          set_number_from_boolean(*new_expr*.*data*.*n*, *mp_true_code*);
        else *set_number_from_boolean*(*new_expr*.*data*.*n*, *mp_false_code*);
        mp_flush_cur_exp(*mp*, *new_expr*);
        mp→*cur_exp*.*type* = *mp_boolean_type*;
      }
#define *type_test*(*A*)
      {
        if (*mp*→*cur_exp*.*type* ≡ (*mp_variable_type*)(*A*))
          set_number_from_boolean(*new_expr*.*data*.*n*, *mp_true_code*);
        else *set_number_from_boolean*(*new_expr*.*data*.*n*, *mp_false_code*);
        mp_flush_cur_exp(*mp*, *new_expr*);
        mp→*cur_exp*.*type* = *mp_boolean_type*;
      }
⟨ Declare unary action procedures 960 ⟩;
static void *mp_do_unary*(**MP** *mp*, **quarterword** *c*)
{
  mp_node *p*;   /∗ for list manipulation ∗/
  mp_value *new_expr*;
  check_arith();
  if (*number_greater*(*internal_value*(*mp_tracing_commands*), *two_t*)) {
     /∗ Trace the current unary operation ∗/
    mp_begin_diagnostic(*mp*);
    mp_print_nl(*mp*, "{");
    mp_print_op(*mp*, *c*);
    mp_print_char(*mp*, *xord*('('));
    mp_print_exp(*mp*, Λ, 0);   /∗ show the operand, but not verbosely ∗/
    mp_print(*mp*, ")}");
    mp_end_diagnostic(*mp*, *false*);
  }
  switch (*c*) {
  case *mp_plus*:
    if (*mp*→*cur_exp*.*type* < *mp_color_type*) *mp_bad_unary*(*mp*, *mp_plus*);
    break;
  case *mp_minus*: *negate_cur_expr*(*mp*);
    break;

```
case mp_not_op:
  if (mp→cur_exp.type ≠ mp_boolean_type) {
    mp_bad_unary(mp, mp_not_op);
  }
  else {
    halfword bb;
    if (cur_exp_value_boolean() ≡ mp_true_code) bb = mp_false_code;
    else  bb = mp_true_code;
    set_cur_exp_value_boolean(bb);
  }
  break;
case mp_sqrt_op: case mp_m_exp_op: case mp_m_log_op: case mp_sin_d_op: case mp_cos_d_op:
  case mp_floor_op: case mp_uniform_deviate: case mp_odd_op: case mp_char_exists_op:
  if (mp→cur_exp.type ≠ mp_known) {
    mp_bad_unary(mp, c);
  }
  else {
    switch (c) {
    case mp_sqrt_op:
      {
        mp_number r1;
        new_number(r1);
        square_rt(r1, cur_exp_value_number());
        set_cur_exp_value_number(r1);
        free_number(r1);
      }
      break;
    case mp_m_exp_op:
      {
        mp_number r1;
        new_number(r1);
        m_exp(r1, cur_exp_value_number());
        set_cur_exp_value_number(r1);
        free_number(r1);
      }
      break;
    case mp_m_log_op:
      {
        mp_number r1;
        new_number(r1);
        m_log(r1, cur_exp_value_number());
        set_cur_exp_value_number(r1);
        free_number(r1);
      }
      break;
    case mp_sin_d_op: case mp_cos_d_op:
      {
        mp_number n_sin, n_cos, arg1, arg2;
        new_number(arg1);
        new_number(arg2);
        new_fraction(n_sin);
```

```
    new_fraction(n_cos);       /* results computed by n_sin_cos */
    number_clone(arg1, cur_exp_value_number());
    number_clone(arg2, unity_t);
    number_multiply_int(arg2, 360);
    number_modulo(arg1, arg2);
    convert_scaled_to_angle(arg1);
    n_sin_cos(arg1, n_cos, n_sin);
    if (c ≡ mp_sin_d_op) {
      fraction_to_round_scaled(n_sin);
      set_cur_exp_value_number(n_sin);
    }
    else {
      fraction_to_round_scaled(n_cos);
      set_cur_exp_value_number(n_cos);
    }
    free_number(arg1);
    free_number(arg2);
    free_number(n_sin);
    free_number(n_cos);
  }
  break;
case mp_floor_op:
  {
    mp_number vvx;

    new_number(vvx);
    number_clone(vvx, cur_exp_value_number());
    floor_scaled(vvx);
    set_cur_exp_value_number(vvx);
    free_number(vvx);
  }
  break;
case mp_uniform_deviate:
  {
    mp_number vvx;

    new_number(vvx);
    mp_unif_rand(mp, &vvx, cur_exp_value_number());
    set_cur_exp_value_number(vvx);
    free_number(vvx);
  }
  break;
case mp_odd_op:
  {
    integer vvx = odd(round_unscaled(cur_exp_value_number()));

    boolean_reset(vvx);
    mp→cur_exp.type = mp_boolean_type;
  }
  break;
case mp_char_exists_op:      /* Determine if a character has been shipped out */
  set_cur_exp_value_scaled(round_unscaled(cur_exp_value_number()) % 256);
  if (number_negative(cur_exp_value_number())) {
    halfword vv = number_to_scaled(cur_exp_value_number());
```

```
              set_cur_exp_value_scaled(vv + 256);
           }
           boolean_reset(mp→char_exists[number_to_scaled(cur_exp_value_number())]);
           mp→cur_exp.type = mp_boolean_type;
           break;
      }    /* there are no other cases */
   }
   break;
case mp_angle_op:
   if (mp_nice_pair(mp, cur_exp_node(), mp→cur_exp.type)) {
      mp_number narg;

      memset(&new_expr, 0, sizeof(mp_value));
      new_number(new_expr.data.n);
      new_angle(narg);
      p = value_node(cur_exp_node());
      n_arg(narg, value_number(x_part(p)), value_number(y_part(p)));
      number_clone(new_expr.data.n, narg);
      convert_angle_to_scaled(new_expr.data.n);
      free_number(narg);
      mp_flush_cur_exp(mp, new_expr);
   }
   else {
      mp_bad_unary(mp, mp_angle_op);
   }
   break;
case mp_x_part: case mp_y_part:
   if ((mp→cur_exp.type ≡ mp_pair_type) ∨ (mp→cur_exp.type ≡ mp_transform_type))
      mp_take_part(mp, c);
   else if (mp→cur_exp.type ≡ mp_picture_type) mp_take_pict_part(mp, c);
   else mp_bad_unary(mp, c);
   break;
case mp_xx_part: case mp_xy_part: case mp_yx_part: case mp_yy_part:
   if (mp→cur_exp.type ≡ mp_transform_type) mp_take_part(mp, c);
   else if (mp→cur_exp.type ≡ mp_picture_type) mp_take_pict_part(mp, c);
   else mp_bad_unary(mp, c);
   break;
case mp_red_part: case mp_green_part: case mp_blue_part:
   if (mp→cur_exp.type ≡ mp_color_type) mp_take_part(mp, c);
   else if (mp→cur_exp.type ≡ mp_picture_type) {
      if pict_color_type (mp_rgb_model)mp_take_pict_part(mp, c);
      else mp_bad_color_part(mp, c);
   }
   else mp_bad_unary(mp, c);
   break;
case mp_cyan_part: case mp_magenta_part: case mp_yellow_part: case mp_black_part:
   if (mp→cur_exp.type ≡ mp_cmykcolor_type) mp_take_part(mp, c);
   else if (mp→cur_exp.type ≡ mp_picture_type) {
      if pict_color_type (mp_cmyk_model)mp_take_pict_part(mp, c);
      else mp_bad_color_part(mp, c);
   }
   else mp_bad_unary(mp, c);
   break;
```

```
case mp_grey_part:
  if (mp→cur_exp.type ≡ mp_known) ;
  else if (mp→cur_exp.type ≡ mp_picture_type) {
    if pict_color_type (mp_grey_model)mp_take_pict_part(mp, c);
    else mp_bad_color_part(mp, c);
  }
  else mp_bad_unary(mp, c);
  break;
case mp_color_model_part:
  if (mp→cur_exp.type ≡ mp_picture_type) mp_take_pict_part(mp, c);
  else mp_bad_unary(mp, c);
  break;
case mp_font_part: case mp_text_part: case mp_path_part: case mp_pen_part: case mp_dash_part:
    case mp_prescript_part: case mp_postscript_part:
  if (mp→cur_exp.type ≡ mp_picture_type) mp_take_pict_part(mp, c);
  else mp_bad_unary(mp, c);
  break;
case mp_char_op:
  if (mp→cur_exp.type ≠ mp_known) {
    mp_bad_unary(mp, mp_char_op);
  }
  else {
    int vv = round_unscaled(cur_exp_value_number( )) % 256;
    set_cur_exp_value_scaled(vv);
    mp→cur_exp.type = mp_string_type;
    if (number_negative(cur_exp_value_number( ))) {
      vv = number_to_scaled(cur_exp_value_number( )) + 256;
      set_cur_exp_value_scaled(vv);
    }
    {
      unsigned char ss[2];
      ss[0] = (unsigned char) number_to_scaled(cur_exp_value_number( ));
      ss[1] = '\0';
      set_cur_exp_str(mp_rtsl(mp, (char *) ss, 1));
    }
  }
  break;
case mp_decimal:
  if (mp→cur_exp.type ≠ mp_known) {
    mp_bad_unary(mp, mp_decimal);
  }
  else {
    mp→old_setting = mp→selector;
    mp→selector = new_string;
    print_number(cur_exp_value_number( ));
    set_cur_exp_str(mp_make_string(mp));
    mp→selector = mp→old_setting;
    mp→cur_exp.type = mp_string_type;
  }
  break;
case mp_oct_op: case mp_hex_op: case mp_ASCII_op:
  if (mp→cur_exp.type ≠ mp_string_type) mp_bad_unary(mp, c);
```

  else $mp\_str\_to\_num(mp, c)$;
  break;
 case mp_font_size:
  if $(mp\text{-}cur\_exp.type \neq mp\_string\_type)$ {
   $mp\_bad\_unary(mp, mp\_font\_size)$;
  }
  else { /∗ Find the design size of the font whose name is cur_exp ∗/
    /∗ One simple application of $find\_font$ is the implementation of the $font\_size$ operator that gets
     the design size for a given font name. ∗/
   $memset(\&new\_expr, 0, \textbf{sizeof}(\textbf{mp\_value}))$;
   $new\_number(new\_expr.data.n)$;
   $set\_number\_from\_scaled(new\_expr.data.n, (mp\text{-}font\_dsize[mp\_find\_font(mp, mp\_str(mp,$
    $cur\_exp\_str())))] + 8)/16)$;
   $mp\_flush\_cur\_exp(mp, new\_expr)$;
  }
  break;
 case mp_length_op: /∗ The length operation is somewhat unusual in that it applies to a variety of
    different types of operands. ∗/
  switch $(mp\text{-}cur\_exp.type)$ {
  case mp_string_type: $memset(\&new\_expr, 0, \textbf{sizeof}(\textbf{mp\_value}))$;
   $new\_number(new\_expr.data.n)$;
   $number\_clone(new\_expr.data.n, unity\_t)$;
   $number\_multiply\_int(new\_expr.data.n, cur\_exp\_str()\text{-}len)$;
   $mp\_flush\_cur\_exp(mp, new\_expr)$;
   break;
  case mp_path_type: $memset(\&new\_expr, 0, \textbf{sizeof}(\textbf{mp\_value}))$;
   $new\_number(new\_expr.data.n)$;
   $mp\_path\_length(mp, \&new\_expr.data.n)$;
   $mp\_flush\_cur\_exp(mp, new\_expr)$;
   break;
  case mp_known: $set\_cur\_exp\_value\_number(cur\_exp\_value\_number())$;
   $number\_abs(cur\_exp\_value\_number())$;
   break;
  case $mp\_picture\_type$: $memset(\&new\_expr, 0, \textbf{sizeof}(\textbf{mp\_value}))$;
   $new\_number(new\_expr.data.n)$;
   $mp\_pict\_length(mp, \&new\_expr.data.n)$;
   $mp\_flush\_cur\_exp(mp, new\_expr)$;
   break;
  default:
   if $(mp\_nice\_pair(mp, cur\_exp\_node(), mp\text{-}cur\_exp.type))$ {
    $memset(\&new\_expr, 0, \textbf{sizeof}(\textbf{mp\_value}))$;
    $new\_number(new\_expr.data.n)$;
    $pyth\_add(new\_expr.data.n, value\_number(x\_part(value\_node(cur\_exp\_node()))),$
     $value\_number(y\_part(value\_node(cur\_exp\_node()))))$;
    $mp\_flush\_cur\_exp(mp, new\_expr)$;
   }
   else $mp\_bad\_unary(mp, c)$;
   break;
  }
  break;
 case $mp\_turning\_op$:
  if $(mp\text{-}cur\_exp.type \equiv mp\_pair\_type)$ {

```
      memset(&new_expr, 0, sizeof(mp_value));
      new_number(new_expr.data.n);
      set_number_to_zero(new_expr.data.n);
      mp_flush_cur_exp(mp, new_expr);
    }
  else if (mp→cur_exp.type ≠ mp_path_type) {
    mp_bad_unary(mp, mp_turning_op);
    }
  else if (mp_left_type(cur_exp_knot()) ≡ mp_endpoint) {
    memset(&new_expr, 0, sizeof(mp_value));
    new_number(new_expr.data.n);
    new_expr.data.p = Λ;
    mp_flush_cur_exp(mp, new_expr);        /* not a cyclic path */
    }
  else {
    memset(&new_expr, 0, sizeof(mp_value));
    new_number(new_expr.data.n);
    mp_turn_cycles_wrapper(mp, &new_expr.data.n, cur_exp_knot());
    mp_flush_cur_exp(mp, new_expr);
    }
  break;
case mp_boolean_type: memset(&new_expr, 0, sizeof(mp_value));
  new_number(new_expr.data.n);
  type_range(mp_boolean_type, mp_unknown_boolean);
  break;
case mp_string_type: memset(&new_expr, 0, sizeof(mp_value));
  new_number(new_expr.data.n);
  type_range(mp_string_type, mp_unknown_string);
  break;
case mp_pen_type: memset(&new_expr, 0, sizeof(mp_value));
  new_number(new_expr.data.n);
  type_range(mp_pen_type, mp_unknown_pen);
  break;
case mp_path_type: memset(&new_expr, 0, sizeof(mp_value));
  new_number(new_expr.data.n);
  type_range(mp_path_type, mp_unknown_path);
  break;
case mp_picture_type: memset(&new_expr, 0, sizeof(mp_value));
  new_number(new_expr.data.n);
  type_range(mp_picture_type, mp_unknown_picture);
  break;
case mp_transform_type: case mp_color_type: case mp_cmykcolor_type: case mp_pair_type:
  memset(&new_expr, 0, sizeof(mp_value));
  new_number(new_expr.data.n);
  type_test(c);
  break;
case mp_numeric_type: memset(&new_expr, 0, sizeof(mp_value));
  new_number(new_expr.data.n);
  type_range(mp_known, mp_independent);
  break;
case mp_known_op: case mp_unknown_op: mp_test_known(mp, c);
  break;
```

case *mp_cycle_op*: *memset*(&*new_expr*, 0, **sizeof**(**mp_value**));
  new_number(*new_expr.data.n*);
  if (*mp⃗cur_exp.type* ≠ *mp_path_type*) *set_number_from_boolean*(*new_expr.data.n*, *mp_false_code*);
  else if (*mp_left_type*(*cur_exp_knot*( )) ≠ *mp_endpoint*)
    set_number_from_boolean(*new_expr.data.n*, *mp_true_code*);
  else *set_number_from_boolean*(*new_expr.data.n*, *mp_false_code*);
  mp_flush_cur_exp(*mp*, *new_expr*);
  mp⃗cur_exp.type = *mp_boolean_type*;
  break;
case *mp_arc_length*:
  if (*mp⃗cur_exp.type* ≡ *mp_pair_type*) *mp_pair_to_path*(*mp*);
  if (*mp⃗cur_exp.type* ≠ *mp_path_type*) {
    mp_bad_unary(*mp*, *mp_arc_length*);
  }
  else {
    memset(&*new_expr*, 0, **sizeof**(**mp_value**));
    new_number(*new_expr.data.n*);
    mp_get_arc_length(*mp*, &*new_expr.data.n*, *cur_exp_knot*( ));
    mp_flush_cur_exp(*mp*, *new_expr*);
  }
  break;
case *mp_filled_op*: **case** *mp_stroked_op*: **case** *mp_textual_op*: **case** *mp_clipped_op*:
  case *mp_bounded_op*:
    /∗ Here we use the fact that *c* − *filled_op* + *fill_code* is the desired graphical object *type*. ∗/
    memset(&*new_expr*, 0, **sizeof**(**mp_value**));
    new_number(*new_expr.data.n*);
    if (*mp⃗cur_exp.type* ≠ *mp_picture_type*) {
      set_number_from_boolean(*new_expr.data.n*, *mp_false_code*);
    }
    else if (*mp_link*(*edge_list*(*cur_exp_node*( ))) ≡ Λ) {
      set_number_from_boolean(*new_expr.data.n*, *mp_false_code*);
    }
    else if (*mp_type*(*mp_link*(*edge_list*(*cur_exp_node*( )))) ≡ (*mp_variable_type*)(*c* + *mp_fill_node_type* −
        mp_filled_op)) {
      set_number_from_boolean(*new_expr.data.n*, *mp_true_code*);
    }
    else {
      set_number_from_boolean(*new_expr.data.n*, *mp_false_code*);
    }
    mp_flush_cur_exp(*mp*, *new_expr*);
    mp⃗cur_exp.type = *mp_boolean_type*;
    break;
case *mp_make_pen_op*:
  if (*mp⃗cur_exp.type* ≡ *mp_pair_type*) *mp_pair_to_path*(*mp*);
  if (*mp⃗cur_exp.type* ≠ *mp_path_type*) *mp_bad_unary*(*mp*, *mp_make_pen_op*);
  else {
    mp⃗cur_exp.type = *mp_pen_type*;
    set_cur_exp_knot(*mp_make_pen*(*mp*, *cur_exp_knot*( ), *true*));
  }
  break;
case *mp_make_path_op*:
  if (*mp⃗cur_exp.type* ≠ *mp_pen_type*) {

```
      mp_bad_unary(mp, mp_make_path_op);
    }
    else {
      mp→cur_exp.type = mp_path_type;
      mp_make_path(mp, cur_exp_knot());
    }
    break;
  case mp_reverse:
    if (mp→cur_exp.type ≡ mp_path_type) {
      mp_knot pk = mp_htap_ypoc(mp, cur_exp_knot());

      if (mp_right_type(pk) ≡ mp_endpoint)  pk = mp_next_knot(pk);
      mp_toss_knot_list(mp, cur_exp_knot());
      set_cur_exp_knot(pk);
    }
    else if (mp→cur_exp.type ≡ mp_pair_type) {
      mp_pair_to_path(mp);
    }
    else {
      mp_bad_unary(mp, mp_reverse);
    }
    break;
  case mp_ll_corner_op:
    if (¬mp_get_cur_bbox(mp))  mp_bad_unary(mp, mp_ll_corner_op);
    else  mp_pair_value(mp, mp_minx, mp_miny);
    break;
  case mp_lr_corner_op:
    if (¬mp_get_cur_bbox(mp))  mp_bad_unary(mp, mp_lr_corner_op);
    else  mp_pair_value(mp, mp_maxx, mp_miny);
    break;
  case mp_ul_corner_op:
    if (¬mp_get_cur_bbox(mp))  mp_bad_unary(mp, mp_ul_corner_op);
    else  mp_pair_value(mp, mp_minx, mp_maxy);
    break;
  case mp_ur_corner_op:
    if (¬mp_get_cur_bbox(mp))  mp_bad_unary(mp, mp_ur_corner_op);
    else  mp_pair_value(mp, mp_maxx, mp_maxy);
    break;
  case mp_read_from_op: case mp_close_from_op:
    if (mp→cur_exp.type ≠ mp_string_type)  mp_bad_unary(mp, c);
    else  mp_do_read_or_close(mp, c);
    break;
  }     /* there are no other cases */
  check_arith();
}
```

960.   The *nice_pair* function returns *true* if both components of a pair are known.

⟨ Declare unary action procedures 960 ⟩ ≡
  static boolean *mp_nice_pair*(**MP** *mp*, **mp_node** *p*, **quarterword** *t*)
  {
    (**void**) *mp*;
    if (*t* ≡ *mp_pair_type*) {
      p = *value_node*(*p*);
      if (*mp_type*(*x_part*(*p*)) ≡ *mp_known*)
        if (*mp_type*(*y_part*(*p*)) ≡ *mp_known*) **return** *true*;
    }
    return *false*;
  }

See also sections 961, 962, 963, 964, 965, 966, 969, 973, 974, 975, 976, 977, 978, 980, 981, 982, 983, 984, and 985.

This code is used in section 959.

961.   The *nice_color_or_pair* function is analogous except that it also accepts fully known colors.

⟨ Declare unary action procedures 960 ⟩ +≡
  static boolean *mp_nice_color_or_pair*(**MP** *mp*, **mp_node** *p*, **quarterword** *t*)
  {
    mp_node *q*;
    (**void**) *mp*;
    switch (*t*) {
    case *mp_pair_type*: *q* = *value_node*(*p*);
      if (*mp_type*(*x_part*(*q*)) ≡ *mp_known*)
        if (*mp_type*(*y_part*(*q*)) ≡ *mp_known*) **return** *true*;
      break;
    case *mp_color_type*: *q* = *value_node*(*p*);
      if (*mp_type*(*red_part*(*q*)) ≡ *mp_known*)
        if (*mp_type*(*green_part*(*q*)) ≡ *mp_known*)
          if (*mp_type*(*blue_part*(*q*)) ≡ *mp_known*) **return** *true*;
      break;
    case *mp_cmykcolor_type*: *q* = *value_node*(*p*);
      if (*mp_type*(*cyan_part*(*q*)) ≡ *mp_known*)
        if (*mp_type*(*magenta_part*(*q*)) ≡ *mp_known*)
          if (*mp_type*(*yellow_part*(*q*)) ≡ *mp_known*)
            if (*mp_type*(*black_part*(*q*)) ≡ *mp_known*) **return** *true*;
      break;
    }
    return *false*;
  }

962.   ⟨Declare unary action procedures 960⟩ +≡
  static void *mp_print_known_or_unknown_type*(**MP** *mp*, **quarterword** *t*, **mp_node** *v*)
  {
      mp_print_char(*mp*, *xord*('('));
      if (*t* > *mp_known*) *mp_print*(*mp*, "unknown␣numeric");
      else {
         if ((*t* ≡ *mp_pair_type*) ∨ (*t* ≡ *mp_color_type*) ∨ (*t* ≡ *mp_cmykcolor_type*))
            if (¬*mp_nice_color_or_pair*(*mp*, *v*, *t*)) *mp_print*(*mp*, "unknown␣");
         mp_print_type(*mp*, *t*);
      }
      mp_print_char(*mp*, *xord*(')'));
  }

963.   ⟨Declare unary action procedures 960⟩ +≡
  static void *mp_bad_unary*(**MP** *mp*, **quarterword** *c*)
  {
      char *msg*[256];
      mp_string *sname*;
      int *old_setting* = *mp*→*selector*;
      const char *∗hlp*[] = {"I'm␣afraid␣I␣don't␣know␣how␣to␣apply␣that␣operation␣to␣that",
         "particular␣type.␣Continue,␣and␣I'll␣simply␣return␣the",
         "argument␣(shown␣above)␣as␣the␣result␣of␣the␣operation.", Λ};
      mp→*selector* = *new_string*;
      mp_print_op(*mp*, *c*);
      mp_print_known_or_unknown_type(*mp*, *mp*→*cur_exp.type*, *cur_exp_node*( ));
      sname = *mp_make_string*(*mp*);
      mp→*selector* = *old_setting*;
      mp_snprintf(*msg*, 256, "Not␣implemented:␣%s", *mp_str*(*mp*, *sname*));
      delete_str_ref(*sname*);
      mp_disp_err(*mp*, Λ);
      mp_back_error(*mp*, *msg*, *hlp*, *true*);
      ;
      mp_get_x_next(*mp*);
  }

964.   Negation is easy except when the current expression is of type *independent*, or when it is a pair with
one or more *independent* components.

⟨Declare unary action procedures 960⟩ +≡
  static void *mp_negate_dep_list*(**MP** *mp*, **mp_value_node** *p*)
  {
      (**void**) *mp*;
      while (1) {
         number_negate(*dep_value*(*p*));
         if (*dep_info*(*p*) ≡ Λ) **return**;
         p = (**mp_value_node**) *mp_link*(*p*);
      }
  }

965.   It is tempting to argue that the negative of an independent variable is an independent variable, hence we don't have to do anything when negating it. The fallacy is that other dependent variables pointing to the current expression must change the sign of their coefficients if we make no change to the current expression.

   Instead, we work around the problem by copying the current expression and recycling it afterwards (cf. the *stash_in* routine).

#define   *negate_value*(A)
   if (*mp_type*(A) ≡ *mp_known*) {
    set_value_number(A, (*value_number*(A)));   /∗ to clear the rest ∗/
    number_negate(*value_number*(A));
   }
   else {
    mp_negate_dep_list(*mp*, (**mp_value_node**) *dep_list*((**mp_value_node**) A));
   }
⟨ Declare unary action procedures 960 ⟩ +≡
 static void *negate_cur_expr*(**MP** *mp*)
 {
  mp_node *p*, *q*, *r*;   /∗ for list manipulation ∗/
  switch (*mp*→*cur_exp.type*) {
  case *mp_color_type*: **case** *mp_cmykcolor_type*: **case** *mp_pair_type*: **case** *mp_independent*:
   q = *cur_exp_node*( );
   mp_make_exp_copy(*mp*, *q*);
   if (*mp*→*cur_exp.type* ≡ *mp_dependent*) {
    mp_negate_dep_list(*mp*, (**mp_value_node**) *dep_list*((**mp_value_node**) *cur_exp_node*( )));
   }
   else if (*mp*→*cur_exp.type* ≤ *mp_pair_type*) {
     /∗ *mp_color_type* *mp_cmykcolor_type*, or *mp_pair_type* ∗/
    p = *value_node*(*cur_exp_node*( ));
    switch (*mp*→*cur_exp.type*) {
    case *mp_pair_type*: *r* = *x_part*(*p*);
     negate_value(*r*);
     r = *y_part*(*p*);
     negate_value(*r*);
     break;
    case *mp_color_type*: *r* = *red_part*(*p*);
     negate_value(*r*);
     r = *green_part*(*p*);
     negate_value(*r*);
     r = *blue_part*(*p*);
     negate_value(*r*);
     break;
    case *mp_cmykcolor_type*: *r* = *cyan_part*(*p*);
     negate_value(*r*);
     r = *magenta_part*(*p*);
     negate_value(*r*);
     r = *yellow_part*(*p*);
     negate_value(*r*);
     r = *black_part*(*p*);
     negate_value(*r*);
     break;
    default: /∗ there are no other valid cases, but please the compiler ∗/
     break;

```
        }
    }        /* if cur_type = mp_known then cur_exp = 0 */
    mp_recycle_value(mp, q);
    mp_free_value_node(mp, q);
    break;
  case mp_dependent: case mp_proto_dependent:
    mp_negate_dep_list(mp, (mp_value_node) dep_list((mp_value_node) cur_exp_node()));
    break;
  case mp_known:
    if (is_number(cur_exp_value_number())) number_negate(cur_exp_value_number());
    break;
  default: mp_bad_unary(mp, mp_minus);
    break;
  }
}
```

966.   If the current expression is a pair, but the context wants it to be a path, we call *pair_to_path*.

⟨ Declare unary action procedures 960 ⟩ +≡

```
static void mp_pair_to_path(MP mp)
{
  set_cur_exp_knot(mp_pair_to_knot(mp));
  mp→cur_exp.type = mp_path_type;
}
```

967.   ⟨ Declarations 8 ⟩ +≡

```
static void mp_bad_color_part(MP mp, quarterword c);
```

968.   **static void** $mp\_bad\_color\_part$(**MP** mp, **quarterword** c)
$\{$
   mp_node p; $/*$ the big node $*/$
   mp_value new_expr;
   char $msg[256]$;
   int $old\_setting$;
   mp_string $sname$;
   const char $*hlp[\,] = \{$"You␣can␣only␣ask␣for␣the␣redpart,␣greenpart,␣bluepart␣of␣a␣$\backslash$
      rgb␣object,",
      "the␣cyanpart,␣magentapart,␣yellowpart␣or␣blackpart␣of␣a␣cmyk␣object,␣",
      "or␣the␣greypart␣of␣a␣grey␣object.␣No␣mixing␣and␣matching,␣please.", $\Lambda\}$;
   $memset(\&new\_expr, 0, \mathbf{sizeof}(\mathbf{mp\_value}))$;
   $new\_number(new\_expr.data.n)$;
   $p = mp\_link(edge\_list(cur\_exp\_node(\,)))$;
   $mp\_disp\_err(mp, \Lambda)$;
   $old\_setting = mp{\rightarrow}selector$;
   $mp{\rightarrow}selector = new\_string$;
   $mp\_print\_op(mp, c)$;
   $sname = mp\_make\_string(mp)$;
   $mp{\rightarrow}selector = old\_setting$;
   ;
   if $(\mathbf{mp\_color\_model}(p) \equiv mp\_grey\_model)$
      $mp\_snprintf(msg, 256,$ "Wrong␣picture␣color␣model:␣%s␣of␣grey␣object", $mp\_str(mp, sname))$;
   else if $(\mathbf{mp\_color\_model}(p) \equiv mp\_cmyk\_model)$
      $mp\_snprintf(msg, 256,$ "Wrong␣picture␣color␣model:␣%s␣of␣cmyk␣object", $mp\_str(mp, sname))$;
   else if $(\mathbf{mp\_color\_model}(p) \equiv mp\_rgb\_model)$
      $mp\_snprintf(msg, 256,$ "Wrong␣picture␣color␣model:␣%s␣of␣rgb␣object", $mp\_str(mp, sname))$;
   else if $(\mathbf{mp\_color\_model}(p) \equiv mp\_no\_model)$ $mp\_snprintf(msg, 256,$
      "Wrong␣picture␣color␣model:␣%s␣of␣marking␣object", $mp\_str(mp, sname))$;
   else $mp\_snprintf(msg, 256,$ "Wrong␣picture␣color␣model:␣%s␣of␣defaulted␣object", $mp\_str(mp,$
      $sname))$;
   $delete\_str\_ref(sname)$;
   $mp\_error(mp, msg, hlp, true)$;
   if $(c \equiv mp\_black\_part)$ $number\_clone(new\_expr.data.n, unity\_t)$;
   else $set\_number\_to\_zero(new\_expr.data.n)$;
   $mp\_flush\_cur\_exp(mp, new\_expr)$;
$\}$

969.   In the following procedure, *cur_exp* points to a capsule, which points to a big node. We want to delete all but one part of the big node.

⟨ Declare unary action procedures 960 ⟩ +≡
    static void *mp_take_part*(**MP** *mp*, **quarterword** *c*)
    {
        mp_node *p*;   /∗ the big node ∗/
        p = *value_node*(*cur_exp_node*( ));
        set_value_node(*mp→temp_val*, *p*);
        mp_type(*mp→temp_val*) = *mp→cur_exp.type*;
        mp_link(*p*) = *mp→temp_val*;
        mp_free_value_node(*mp*, *cur_exp_node*( ));
        switch (*c*) {
        case *mp_x_part*:
            if (*mp→cur_exp.type* ≡ *mp_pair_type*) *mp_make_exp_copy*(*mp*, *x_part*(*p*));
            else *mp_make_exp_copy*(*mp*, *tx_part*(*p*));
            break;
        case *mp_y_part*:
            if (*mp→cur_exp.type* ≡ *mp_pair_type*) *mp_make_exp_copy*(*mp*, *y_part*(*p*));
            else *mp_make_exp_copy*(*mp*, *ty_part*(*p*));
            break;
        case *mp_xx_part*: *mp_make_exp_copy*(*mp*, *xx_part*(*p*));
            break;
        case *mp_xy_part*: *mp_make_exp_copy*(*mp*, *xy_part*(*p*));
            break;
        case *mp_yx_part*: *mp_make_exp_copy*(*mp*, *yx_part*(*p*));
            break;
        case *mp_yy_part*: *mp_make_exp_copy*(*mp*, *yy_part*(*p*));
            break;
        case *mp_red_part*: *mp_make_exp_copy*(*mp*, *red_part*(*p*));
            break;
        case *mp_green_part*: *mp_make_exp_copy*(*mp*, *green_part*(*p*));
            break;
        case *mp_blue_part*: *mp_make_exp_copy*(*mp*, *blue_part*(*p*));
            break;
        case *mp_cyan_part*: *mp_make_exp_copy*(*mp*, *cyan_part*(*p*));
            break;
        case *mp_magenta_part*: *mp_make_exp_copy*(*mp*, *magenta_part*(*p*));
            break;
        case *mp_yellow_part*: *mp_make_exp_copy*(*mp*, *yellow_part*(*p*));
            break;
        case *mp_black_part*: *mp_make_exp_copy*(*mp*, *black_part*(*p*));
            break;
        }
        mp_recycle_value(*mp*, *mp→temp_val*);
    }

970.   ⟨ Initialize table entries 182 ⟩ +≡
    mp→temp_val = *mp_get_value_node*(*mp*);
    mp_name_type(*mp→temp_val*) = *mp_capsule*;

971.   ⟨ Free table entries 183 ⟩ +≡
   $mp\_free\_value\_node(mp, mp \rightarrow temp\_val)$;

972.   ⟨ Declarations 8 ⟩ +≡
   static mp_edge_header_node mp_scale_edges(**MP** mp, **mp_number** se_sf, **mp_edge_header_node**
      se_pic);

973.   ⟨Declare unary action procedures 960⟩ +≡

  static void *mp_take_pict_part*(**MP** *mp*, **quarterword** *c*)
  {
    mp_node *p*;   /* first graphical object in *cur_exp* */
    mp_value *new_expr*;

    memset(&*new_expr*, 0, **sizeof**(**mp_value**));
    new_number(*new_expr*.*data*.*n*);
    p = *mp_link*(*edge_list*(*cur_exp_node*()));
    if (*p* ≠ Λ) {
      switch (*c*) {
      case *mp_x_part*: **case** *mp_y_part*: **case** *mp_xx_part*: **case** *mp_xy_part*: **case** *mp_yx_part*:
        case *mp_yy_part*:
        if (*mp_type*(*p*) ≡ *mp_text_node_type*) {
          mp_text_node *p0* = (**mp_text_node**) *p*;

          switch (*c*) {
          case *mp_x_part*: *number_clone*(*new_expr*.*data*.*n*, *p0*→*tx*);
            break;
          case *mp_y_part*: *number_clone*(*new_expr*.*data*.*n*, *p0*→*ty*);
            break;
          case *mp_xx_part*: *number_clone*(*new_expr*.*data*.*n*, *p0*→*txx*);
            break;
          case *mp_xy_part*: *number_clone*(*new_expr*.*data*.*n*, *p0*→*txy*);
            break;
          case *mp_yx_part*: *number_clone*(*new_expr*.*data*.*n*, *p0*→*tyx*);
            break;
          case *mp_yy_part*: *number_clone*(*new_expr*.*data*.*n*, *p0*→*tyy*);
            break;
          }
          mp_flush_cur_exp(*mp*, *new_expr*);
        }
        else goto NOT_FOUND;
        break;
      case *mp_red_part*: **case** *mp_green_part*: **case** *mp_blue_part*:
        if (*has_color*(*p*)) {
          switch (*c*) {
          case *mp_red_part*: *number_clone*(*new_expr*.*data*.*n*, ((**mp_stroked_node**) *p*)→*red*);
            break;
          case *mp_green_part*: *number_clone*(*new_expr*.*data*.*n*, ((**mp_stroked_node**) *p*)→*green*);
            break;
          case *mp_blue_part*: *number_clone*(*new_expr*.*data*.*n*, ((**mp_stroked_node**) *p*)→*blue*);
            break;
          }
          mp_flush_cur_exp(*mp*, *new_expr*);
        }
        else goto NOT_FOUND;
        break;
      case *mp_cyan_part*: **case** *mp_magenta_part*: **case** *mp_yellow_part*: **case** *mp_black_part*:
        if (*has_color*(*p*)) {
          if (**mp_color_model**(*p*) ≡ *mp_uninitialized_model* ∧ *c* ≡ *mp_black_part*) {
            set_number_to_unity(*new_expr*.*data*.*n*);
          }
          else {

```
      switch (c) {
      case mp_cyan_part: number_clone(new_expr.data.n, ((mp_stroked_node) p)→cyan);
        break;
      case mp_magenta_part: number_clone(new_expr.data.n, ((mp_stroked_node) p)→magenta);
        break;
      case mp_yellow_part: number_clone(new_expr.data.n, ((mp_stroked_node) p)→yellow);
        break;
      case mp_black_part: number_clone(new_expr.data.n, ((mp_stroked_node) p)→black);
        break;
      }
    }
    mp_flush_cur_exp(mp, new_expr);
  }
  else goto NOT_FOUND;
  break;
case mp_grey_part:
  if (has_color(p)) {
    number_clone(new_expr.data.n, ((mp_stroked_node) p)→grey);
    mp_flush_cur_exp(mp, new_expr);
  }
  else goto NOT_FOUND;
  break;
case mp_color_model_part:
  if (has_color(p)) {
    if (mp_color_model(p) ≡ mp_uninitialized_model) {
      number_clone(new_expr.data.n, internal_value(mp_default_color_model));
    }
    else {
      number_clone(new_expr.data.n, unity_t);
      number_multiply_int(new_expr.data.n, mp_color_model(p));
    }
    mp_flush_cur_exp(mp, new_expr);
  }
  else goto NOT_FOUND;
  break;
case mp_text_part:
  if (mp_type(p) ≠ mp_text_node_type) goto NOT_FOUND;
  else {
    new_expr.data.str = mp_text_p(p);
    add_str_ref(new_expr.data.str);
    mp_flush_cur_exp(mp, new_expr);
    mp→cur_exp.type = mp_string_type;
  }
  ;
  break;
case mp_prescript_part:
  if (¬has_color(p)) {
    goto NOT_FOUND;
  }
  else {
    if (mp_pre_script(p)) {
      new_expr.data.str = mp_pre_script(p);
```

```
            add_str_ref (new_expr .data.str );
          }
          else {
            new_expr .data.str = mp_rts (mp, "");
          }
          mp_flush_cur_exp (mp, new_expr );
          mp→cur_exp.type = mp_string_type ;
        }
        ;
        break;
      case mp_postscript_part :
        if (¬has_color (p)) {
          goto NOT_FOUND;
        }
        else {
          if (mp_post_script (p)) {
            new_expr .data.str = mp_post_script (p);
            add_str_ref (new_expr .data.str );
          }
          else {
            new_expr .data.str = mp_rts (mp, "");
          }
          mp_flush_cur_exp (mp, new_expr );
          mp→cur_exp.type = mp_string_type ;
        }
        ;
        break;
      case mp_font_part :
        if (mp_type (p) ≠ mp_text_node_type ) goto NOT_FOUND;
        else {
          new_expr .data.str = mp_rts (mp, mp→font_name [mp_font_n (p)]);
          add_str_ref (new_expr .data.str );
          mp_flush_cur_exp (mp, new_expr );
          mp→cur_exp.type = mp_string_type ;
        }
        ;
        break;
      case mp_path_part :
        if (mp_type (p) ≡ mp_text_node_type ) {
          goto NOT_FOUND;
        }
        else if (is_stop (p)) {
          mp_confusion (mp, "pict");
        }
        else {
          new_expr .data.node = Λ;
          switch (mp_type (p)) {
          case mp_fill_node_type : new_expr .data.p = mp_copy_path (mp, mp_path_p ((mp_fill_node) p));
            break;
          case mp_stroked_node_type :
            new_expr .data.p = mp_copy_path (mp, mp_path_p ((mp_stroked_node) p));
            break;
```

```
          case mp_start_bounds_node_type:
            new_expr.data.p = mp_copy_path(mp, mp_path_p((mp_start_bounds_node) p));
            break;
          case mp_start_clip_node_type:
            new_expr.data.p = mp_copy_path(mp, mp_path_p((mp_start_clip_node) p));
            break;
          default: assert(0);
            break;
          }
          mp_flush_cur_exp(mp, new_expr);
          mp→cur_exp.type = mp_path_type;
        }
        break;
      case mp_pen_part:
        if (¬has_pen(p)) {
          goto NOT_FOUND;
        }
        else {
          switch (mp_type(p)) {
          case mp_fill_node_type:
            if (mp_pen_p((mp_fill_node) p) ≡ Λ) goto NOT_FOUND;
            else {
              new_expr.data.p = copy_pen(mp_pen_p((mp_fill_node) p));
              mp_flush_cur_exp(mp, new_expr);
              mp→cur_exp.type = mp_pen_type;
            }
            break;
          case mp_stroked_node_type:
            if (mp_pen_p((mp_stroked_node) p) ≡ Λ) goto NOT_FOUND;
            else {
              new_expr.data.p = copy_pen(mp_pen_p((mp_stroked_node) p));
              mp_flush_cur_exp(mp, new_expr);
              mp→cur_exp.type = mp_pen_type;
            }
            break;
          default: assert(0);
            break;
          }
        }
        break;
      case mp_dash_part:
        if (mp_type(p) ≠ mp_stroked_node_type) {
          goto NOT_FOUND;
        }
        else {
          if (mp_dash_p(p) ≡ Λ) {
            goto NOT_FOUND;
          }
          else {
            add_edge_ref(mp_dash_p(p));
            new_expr.data.node = (mp_node) mp_scale_edges(mp, ((mp_stroked_node)
                p)→dash_scale, (mp_edge_header_node) mp_dash_p(p));
```

```
            mp_flush_cur_exp(mp, new_expr);
            mp→cur_exp.type = mp_picture_type;
          }
        }
      break;
    }    /∗ all cases have been enumerated ∗/
    return;
  }
  ;
NOT_FOUND:    /∗ Convert the current expression to a NULL value appropriate for c ∗/
  switch (c) {
  case mp_text_part: case mp_font_part: case mp_prescript_part: case mp_postscript_part:
    new_expr.data.str = mp_rts(mp, "");
    mp_flush_cur_exp(mp, new_expr);
    mp→cur_exp.type = mp_string_type;
    break;
  case mp_path_part: new_expr.data.p = mp_new_knot(mp);
    mp_flush_cur_exp(mp, new_expr);
    mp_left_type(cur_exp_knot()) = mp_endpoint;
    mp_right_type(cur_exp_knot()) = mp_endpoint;
    mp_next_knot(cur_exp_knot()) = cur_exp_knot();
    set_number_to_zero(cur_exp_knot()→x_coord);
    set_number_to_zero(cur_exp_knot()→y_coord);
    mp_originator(cur_exp_knot()) = mp_metapost_user;
    mp→cur_exp.type = mp_path_type;
    break;
  case mp_pen_part: new_expr.data.p = mp_get_pen_circle(mp, zero_t);
    mp_flush_cur_exp(mp, new_expr);
    mp→cur_exp.type = mp_pen_type;
    break;
  case mp_dash_part: new_expr.data.node = (mp_node) mp_get_edge_header_node(mp);
    mp_flush_cur_exp(mp, new_expr);
    mp_init_edges(mp, (mp_edge_header_node) cur_exp_node());
    mp→cur_exp.type = mp_picture_type;
    break;
  default: set_number_to_zero(new_expr.data.n);
    mp_flush_cur_exp(mp, new_expr);
    break;
  }
}
```

974.   ⟨Declare unary action procedures 960⟩ +≡

```
static void mp_str_to_num(MP mp, quarterword c)
{     /* converts a string to a number */
  integer n;     /* accumulator */
  ASCII_code m;     /* current character */
  unsigned k;     /* index into str_pool */
  int b;     /* radix of conversion */
  boolean bad_char;     /* did the string contain an invalid digit? */
  mp_value new_expr;

  memset(&new_expr, 0, sizeof(mp_value));
  new_number(new_expr.data.n);
  if (c ≡ mp_ASCII_op) {
    if (cur_exp_str()→len ≡ 0)  n = −1;
    else  n = cur_exp_str()→str[0];
  }
  else {
    if (c ≡ mp_oct_op)  b = 8;
    else  b = 16;
    n = 0;
    bad_char = false;
    for (k = 0;  k < cur_exp_str()→len;  k++) {
      m = (ASCII_code)(∗(cur_exp_str()→str + k));
      if ((m ≥ '0') ∧ (m ≤ '9'))  m = (ASCII_code)(m − '0');
      else if ((m ≥ 'A') ∧ (m ≤ 'F'))  m = (ASCII_code)(m − 'A' + 10);
      else if ((m ≥ 'a') ∧ (m ≤ 'f'))  m = (ASCII_code)(m − 'a' + 10);
      else {
        bad_char = true;
        m = 0;
      }
      ;
      if ((int) m ≥ b) {
        bad_char = true;
        m = 0;
      }
      ;
      if (n < 32768/b)  n = n ∗ b + m;
      else  n = 32767;
    }     /* Give error messages if bad_char or n ≥ 4096 */
    if (bad_char) {
      const char ∗hlp[] = {"I␣zeroed␣out␣characters␣that␣weren't␣hex␣digits.", Λ};
      if (c ≡ mp_oct_op) {
        hlp[0] = "I␣zeroed␣out␣characters␣that␣weren't␣in␣the␣range␣0..7.";
      }
      mp_disp_err(mp, Λ);
      mp_back_error(mp, "String␣contains␣illegal␣digits", hlp, true);
      mp_get_x_next(mp);
    }
    if ((n > 4095)) {     /* todo, this is scaled specific */
      if (number_positive(internal_value(mp_warning_check))) {
        char msg[256];
        const char ∗hlp[] = {"I␣have␣trouble␣with␣numbers␣greater␣than␣4095;␣watch␣out.",
            "(Set␣warningcheck:=0␣to␣suppress␣this␣message.)", Λ};
```

```
            mp_snprintf (msg, 256, "Number␣too␣large␣(%d)", (int) n);
            mp_back_error (mp, msg, hlp, true);
            mp_get_x_next (mp);
          }
        }
      }
      number_clone (new_expr.data.n, unity_t);
      number_multiply_int (new_expr.data.n, n);
      mp_flush_cur_exp (mp, new_expr );
    }
```

975. ⟨Declare unary action procedures 960⟩ +≡
```
  static void mp_path_length (MP mp, mp_number ∗n)
  {      /∗ computes the length of the current path ∗/
    mp_knot p;      /∗ traverser ∗/
    set_number_to_zero (∗n);
    p = cur_exp_knot ( );
    if (mp_left_type (p) ≡ mp_endpoint ) {
      number_substract (∗n, unity_t);      /∗ -unity ∗/
    }
    do {
      p = mp_next_knot (p);
      number_add (∗n, unity_t);
    } while (p ≠ cur_exp_knot ( ));
  }
```

976. ⟨Declare unary action procedures 960⟩ +≡
```
  static void mp_pict_length (MP mp, mp_number ∗n)
  {      /∗ counts interior components in picture cur_exp ∗/
    mp_node p;      /∗ traverser ∗/
    set_number_to_zero (∗n);
    p = mp_link (edge_list (cur_exp_node ( )));
    if (p ≠ Λ) {
      if (is_start_or_stop (p))
        if (mp_skip_1component (mp, p) ≡ Λ) p = mp_link (p);
      while (p ≠ Λ) {
        if (¬is_start_or_stop (p)) p = mp_link (p);
        else if (¬is_stop (p)) p = mp_skip_1component (mp, p);
        else return;
        number_add (∗n, unity_t);
      }
    }
  }
```

977. The function *an_angle* returns the value of the *angle* primitive, or 0 if the argument is *origin*.

⟨ Declare unary action procedures 960 ⟩ +≡
  static void *mp_an_angle*(**MP** *mp*, **mp_number** *∗ret*, **mp_number** *xpar*, **mp_number** *ypar*)
  {
    set_number_to_zero(*∗ret*);
    if ((¬(*number_zero*(*xpar*) ∧ *number_zero*(*ypar*)))) {
      n_arg(*∗ret*, *xpar*, *ypar*);
    }
  }

978. The actual turning number is (for the moment) computed in a C function that receives eight integers corresponding to the four controlling points, and returns a single angle. Besides those, we have to account for discrete moves at the actual points.

#define *mp_floor*(*a*) ((*a*) ≥ 0 ? (**int**)(*a*) : −(**int**)(−(*a*)))
#define *bezier_error* (720 ∗ (256 ∗ 256 ∗ 16)) + 1
#define *mp_sign*(*v*) ((*v*) > 0 ? 1 : ((*v*) < 0 ? −1 : 0))
#define *mp_out*(*A*) (**double**)((*A*)/16)

⟨ Declare unary action procedures 960 ⟩ +≡
  static void *mp_bezier_slope*(**MP** *mp*, **mp_number** *∗ret*, **mp_number** AX, **mp_number**
      AY, **mp_number** BX, **mp_number** BY, **mp_number** CX, **mp_number** CY, **mp_number**
      DX, **mp_number** DY);

979.   **static void** *mp_bezier_slope*(**MP** *mp*, **mp_number** *∗ret*, **mp_number** AX, **mp_number**
         AY, **mp_number** BX, **mp_number** BY, **mp_number** CX, **mp_number** CY, **mp_number**
         DX, **mp_number** DY)
  {
     double *a*, *b*, *c*;
     mp_number *deltax*, *deltay*;
     double *ax*, *ay*, *bx*, *by*, *cx*, *cy*, *dx*, *dy*;
     mp_number *xi*, *xo*, *xm*;
     double *res* = 0;
     ax = *number_to_double*(AX);
     ay = *number_to_double*(AY);
     bx = *number_to_double*(BX);
     by = *number_to_double*(BY);
     cx = *number_to_double*(CX);
     cy = *number_to_double*(CY);
     dx = *number_to_double*(DX);
     dy = *number_to_double*(DY);
     new_number(*deltax*);
     new_number(*deltay*);
     set_number_from_substraction(*deltax*, BX, AX);
     set_number_from_substraction(*deltay*, BY, AY);
     if (*number_zero*(*deltax*) ∧ *number_zero*(*deltay*)) {
        set_number_from_substraction(*deltax*, CX, AX);
        set_number_from_substraction(*deltay*, CY, AY);
     }
     if (*number_zero*(*deltax*) ∧ *number_zero*(*deltay*)) {
        set_number_from_substraction(*deltax*, DX, AX);
        set_number_from_substraction(*deltay*, DY, AY);
     }
     new_number(*xi*);
     new_number(*xm*);
     new_number(*xo*);
     mp_an_angle(*mp*, &*xi*, *deltax*, *deltay*);
     set_number_from_substraction(*deltax*, CX, BX);
     set_number_from_substraction(*deltay*, CY, BY);
     mp_an_angle(*mp*, &*xm*, *deltax*, *deltay*);   /∗ !!! never used? ∗/
     set_number_from_substraction(*deltax*, DX, CX);
     set_number_from_substraction(*deltay*, DY, CY);
     if (*number_zero*(*deltax*) ∧ *number_zero*(*deltay*)) {
        set_number_from_substraction(*deltax*, DX, BX);
        set_number_from_substraction(*deltay*, DY, BY);
     }
     if (*number_zero*(*deltax*) ∧ *number_zero*(*deltay*)) {
        set_number_from_substraction(*deltax*, DX, AX);
        set_number_from_substraction(*deltay*, DY, AY);
     }
     mp_an_angle(*mp*, &*xo*, *deltax*, *deltay*);
     a = (*bx* − *ax*) ∗ (*cy* − *by*) − (*cx* − *bx*) ∗ (*by* − *ay*);   /∗ a = (bp-ap)x(cp-bp); ∗/
     b = (*bx* − *ax*) ∗ (*dy* − *cy*) − (*by* − *ay*) ∗ (*dx* − *cx*);
     ;   /∗ b = (bp-ap)x(dp-cp); ∗/
     c = (*cx* − *bx*) ∗ (*dy* − *cy*) − (*dx* − *cx*) ∗ (*cy* − *by*);   /∗ c = (cp-bp)x(dp-cp); ∗/
     if ((*a* ≡ 0) ∧ (*c* ≡ 0)) {

$res = (b \equiv 0\ ?\ 0 : (mp\_out\,(number\_to\_double\,(xo)) - mp\_out\,(number\_to\_double\,(xi))));$
   }
  else if $((a \equiv 0) \vee (c \equiv 0))$ {
    if $((mp\_sign\,(b) \equiv mp\_sign\,(a)) \vee (mp\_sign\,(b) \equiv mp\_sign\,(c)))$ {
      $res = mp\_out\,(number\_to\_double\,(xo)) - mp\_out\,(number\_to\_double\,(xi));$ /* ? */
      if $(res < -180.0)$ $res\mathrel{+}= 360.0;$
      else if $(res > 180.0)$ $res\mathrel{-}= 360.0;$
    }
    else {
      $res = mp\_out\,(number\_to\_double\,(xo)) - mp\_out\,(number\_to\_double\,(xi));$ /* ? */
    }
  }
  else if $((mp\_sign\,(a) * mp\_sign\,(c)) < 0)$ {
    $res = mp\_out\,(number\_to\_double\,(xo)) - mp\_out\,(number\_to\_double\,(xi));$ /* ? */
    if $(res < -180.0)$ $res\mathrel{+}= 360.0;$
    else if $(res > 180.0)$ $res\mathrel{-}= 360.0;$
  }
  else {
    if $(mp\_sign\,(a) \equiv mp\_sign\,(b))$ {
      $res = mp\_out\,(number\_to\_double\,(xo)) - mp\_out\,(number\_to\_double\,(xi));$ /* ? */
      if $(res < -180.0)$ $res\mathrel{+}= 360.0;$
      else if $(res > 180.0)$ $res\mathrel{-}= 360.0;$
    }
    else {
      if $((b * b) \equiv (4 * a * c))$ {
        $res = (\textbf{double})\ bezier\_error\,;$
      }
      else if $((b * b) < (4 * a * c))$ {
        $res = mp\_out\,(number\_to\_double\,(xo)) - mp\_out\,(number\_to\_double\,(xi));$ /* ? */
        if $(res \leq 0.0 \wedge res > -180.0)$ $res\mathrel{+}= 360.0;$
        else if $(res \geq 0.0 \wedge res < 180.0)$ $res\mathrel{-}= 360.0;$
      }
      else {
        $res = mp\_out\,(number\_to\_double\,(xo)) - mp\_out\,(number\_to\_double\,(xi));$
        if $(res < -180.0)$ $res\mathrel{+}= 360.0;$
        else if $(res > 180.0)$ $res\mathrel{-}= 360.0;$
      }
    }
  }
$free\_number\,(deltax\,);$
$free\_number\,(deltay\,);$
$free\_number\,(xi);$
$free\_number\,(xo);$
$free\_number\,(xm);$
$set\_number\_from\_double\,(*ret, res\,);$
$convert\_scaled\_to\_angle\,(*ret\,);$
}

980.

#define $p\_nextnext$ $mp\_next\_knot(mp\_next\_knot(p))$
#define p_next $mp\_next\_knot(p)$

⟨ Declare unary action procedures 960 ⟩ +≡

 static void $mp\_turn\_cycles(\mathbf{MP}\ mp, \mathbf{mp\_number}\ *turns, \mathbf{mp\_knot}\ c)$
 {
    $mp\_angle\ res, ang;$ /∗ the angles of intermediate results ∗/

    mp_knot $p;$ /∗ for running around the path ∗/
    mp_number $xp, yp;$ /∗ coordinates of next point ∗/
    mp_number $x, y;$ /∗ helper coordinates ∗/
    mp_number $arg1, arg2;$

    $mp\_angle\ in\_angle, out\_angle;$ /∗ helper angles ∗/
    $mp\_angle\ seven\_twenty\_deg\_t, neg\_one\_eighty\_deg\_t;$

    unsigned $old\_setting;$ /∗ saved $selector$ setting ∗/

    $set\_number\_to\_zero(*turns);$
    $new\_number(arg1);$
    $new\_number(arg2);$
    $new\_number(xp);$
    $new\_number(yp);$
    $new\_number(x);$
    $new\_number(y);$
    $new\_angle(in\_angle);$
    $new\_angle(out\_angle);$
    $new\_angle(ang);$
    $new\_angle(res);$
    $new\_angle(seven\_twenty\_deg\_t);$
    $new\_angle(neg\_one\_eighty\_deg\_t);$
    $number\_clone(seven\_twenty\_deg\_t, three\_sixty\_deg\_t);$
    $number\_double(seven\_twenty\_deg\_t);$
    $number\_clone(neg\_one\_eighty\_deg\_t, one\_eighty\_deg\_t);$
    $number\_negate(neg\_one\_eighty\_deg\_t);$
    $p = c;$
    $old\_setting = mp{\rightarrow}selector;$
    $mp{\rightarrow}selector = term\_only;$
    if $(number\_greater(internal\_value(mp\_tracing\_commands), unity\_t))$ {
       $mp\_begin\_diagnostic(mp);$
       $mp\_print\_nl(mp, "");$
       $mp\_end\_diagnostic(mp, false);$
    }
    do {
       $number\_clone(xp, p\_next{\rightarrow}x\_coord);$
       $number\_clone(yp, p\_next{\rightarrow}y\_coord);$
       $mp\_bezier\_slope(mp, \&ang, p{\rightarrow}x\_coord, p{\rightarrow}y\_coord, p{\rightarrow}right\_x, p{\rightarrow}right\_y, p\_next{\rightarrow}left\_x, p\_next{\rightarrow}left\_y, xp,$
          $yp);$
       if $(number\_greater(ang, seven\_twenty\_deg\_t))$ {
          $mp\_error(mp, "Strange_path", \Lambda, true);$
          $mp{\rightarrow}selector = old\_setting;$
          $set\_number\_to\_zero(*turns);$
          goto DONE;
       }
       $number\_add(res, ang);$

```
      if (number_greater(res, one_eighty_deg_t)) {
        number_substract(res, three_sixty_deg_t);
        number_add(*turns, unity_t);
      }
      if (number_lessequal(res, neg_one_eighty_deg_t)) {
        number_add(res, three_sixty_deg_t);
        number_substract(*turns, unity_t);
      }     /* incoming angle at next point */
      number_clone(x, p_next→left_x);
      number_clone(y, p_next→left_y);
      if (number_equal(xp, x) ∧ number_equal(yp, y)) {
        number_clone(x, p→right_x);
        number_clone(y, p→right_y);
      }
      if (number_equal(xp, x) ∧ number_equal(yp, y)) {
        number_clone(x, p→x_coord);
        number_clone(y, p→y_coord);
      }
      set_number_from_substraction(arg1, xp, x);
      set_number_from_substraction(arg2, yp, y);
      mp_an_angle(mp, &in_angle, arg1, arg2);      /* outgoing angle at next point */
      number_clone(x, p_next→right_x);
      number_clone(y, p_next→right_y);
      if (number_equal(xp, x) ∧ number_equal(yp, y)) {
        number_clone(x, p_nextnext→left_x);
        number_clone(y, p_nextnext→left_y);
      }
      if (number_equal(xp, x) ∧ number_equal(yp, y)) {
        number_clone(x, p_nextnext→x_coord);
        number_clone(y, p_nextnext→y_coord);
      }
      set_number_from_substraction(arg1, x, xp);
      set_number_from_substraction(arg2, y, yp);
      mp_an_angle(mp, &out_angle, arg1, arg2);
      set_number_from_substraction(ang, out_angle, in_angle);
      mp_reduce_angle(mp, &ang);
      if (number_nonzero(ang)) {
        number_add(res, ang);
        if (number_greaterequal(res, one_eighty_deg_t)) {
          number_substract(res, three_sixty_deg_t);
          number_add(*turns, unity_t);
        }
        if (number_lessequal(res, neg_one_eighty_deg_t)) {
          number_add(res, three_sixty_deg_t);
          number_substract(*turns, unity_t);
        }
      }
    }
    p = mp_next_knot(p);
  } while (p ≠ c);
  mp→selector = old_setting;
DONE: free_number(xp);
  free_number(yp);
```

$free\_number(x);$
$free\_number(y);$
$free\_number(seven\_twenty\_deg\_t);$
$free\_number(neg\_one\_eighty\_deg\_t);$
$free\_number(in\_angle);$
$free\_number(out\_angle);$
$free\_number(ang);$
$free\_number(res);$
$free\_number(arg1);$
$free\_number(arg2);$
}

981.   ⟨Declare unary action procedures 960⟩ +≡
static void $mp\_turn\_cycles\_wrapper(\mathbf{MP}\ mp, \mathbf{mp\_number}\ *ret, \mathbf{mp\_knot}\ c)$
{
  if $(mp\_next\_knot(c) \equiv c)$ {   /∗ one-knot paths always have a turning number of 1 ∗/
    $set\_number\_to\_unity(*ret);$
  }
  else {
    $mp\_turn\_cycles(mp, ret, c);$
  }
}

982. ⟨Declare unary action procedures 960⟩ +≡

```
static void mp_test_known(MP mp, quarterword c)
{
  int b;        /* is the current expression known? */
  mp_node p;      /* location in a big node */
  mp_value new_expr;

  memset(&new_expr, 0, sizeof(mp_value));
  new_number(new_expr.data.n);
  b = mp_false_code;
  switch (mp→cur_exp.type) {
  case mp_vacuous: case mp_boolean_type: case mp_string_type: case mp_pen_type:
    case mp_path_type: case mp_picture_type: case mp_known: b = mp_true_code;
    break;
  case mp_transform_type: p = value_node(cur_exp_node());
    if (mp_type(tx_part(p)) ≠ mp_known) break;
    if (mp_type(ty_part(p)) ≠ mp_known) break;
    if (mp_type(xx_part(p)) ≠ mp_known) break;
    if (mp_type(xy_part(p)) ≠ mp_known) break;
    if (mp_type(yx_part(p)) ≠ mp_known) break;
    if (mp_type(yy_part(p)) ≠ mp_known) break;
    b = mp_true_code;
    break;
  case mp_color_type: p = value_node(cur_exp_node());
    if (mp_type(red_part(p)) ≠ mp_known) break;
    if (mp_type(green_part(p)) ≠ mp_known) break;
    if (mp_type(blue_part(p)) ≠ mp_known) break;
    b = mp_true_code;
    break;
  case mp_cmykcolor_type: p = value_node(cur_exp_node());
    if (mp_type(cyan_part(p)) ≠ mp_known) break;
    if (mp_type(magenta_part(p)) ≠ mp_known) break;
    if (mp_type(yellow_part(p)) ≠ mp_known) break;
    if (mp_type(black_part(p)) ≠ mp_known) break;
    b = mp_true_code;
    break;
  case mp_pair_type: p = value_node(cur_exp_node());
    if (mp_type(x_part(p)) ≠ mp_known) break;
    if (mp_type(y_part(p)) ≠ mp_known) break;
    b = mp_true_code;
    break;
  default: break;
  }
  if (c ≡ mp_known_op) {
    set_number_from_boolean(new_expr.data.n, b);
  }
  else {
    if (b ≡ mp_true_code) {
      set_number_from_boolean(new_expr.data.n, mp_false_code);
    }
    else {
      set_number_from_boolean(new_expr.data.n, mp_true_code);
    }
```

```
  }
  mp_flush_cur_exp(mp, new_expr);
  cur_exp_node() = Λ;      /* !! do not replace with set_cur_exp_node() !! */
  mp→cur_exp.type = mp_boolean_type;
}
```

983.   The *pair_value* routine changes the current expression to a given ordered pair of values.

⟨ Declare unary action procedures 960 ⟩ +≡

```
  static void mp_pair_value(MP mp, mp_number x, mp_number y)
  {
    mp_node p;      /* a pair node */
    mp_value new_expr;
    mp_number x1, y1;
    new_number(x1);
    new_number(y1);
    number_clone(x1, x);
    number_clone(y1, y);
    memset(&new_expr, 0, sizeof(mp_value));
    new_number(new_expr.data.n);
    p = mp_get_value_node(mp);
    new_expr.type = mp_type(p);
    new_expr.data.node = p;
    mp_flush_cur_exp(mp, new_expr);
    mp→cur_exp.type = mp_pair_type;
    mp_name_type(p) = mp_capsule;
    mp_init_pair_node(mp, p);
    p = value_node(p);
    mp_type(x_part(p)) = mp_known;
    set_value_number(x_part(p), x1);
    mp_type(y_part(p)) = mp_known;
    set_value_number(y_part(p), y1);
    free_number(x1);
    free_number(y1);
  }
```

984.   Here is a function that sets *minx*, *maxx*, *miny*, *maxy* to the bounding box of the current expression.
The boolean result is *false* if the expression has the wrong type.

⟨ Declare unary action procedures 960 ⟩ +≡
  static boolean *mp_get_cur_bbox*(**MP** *mp*)
  {
    switch (*mp→cur_exp.type*) {
    case *mp_picture_type*:
      {
        mp_edge_header_node *p0* = (**mp_edge_header_node**) *cur_exp_node*( );

        mp_set_bbox(*mp*, *p0*, *true*);
        if (*number_greater*(*p0→minx*, *p0→maxx*)) {
          set_number_to_zero(*mp_minx*);
          set_number_to_zero(*mp_maxx*);
          set_number_to_zero(*mp_miny*);
          set_number_to_zero(*mp_maxy*);
        }
        else {
          number_clone(*mp_minx*, *p0→minx*);
          number_clone(*mp_maxx*, *p0→maxx*);
          number_clone(*mp_miny*, *p0→miny*);
          number_clone(*mp_maxy*, *p0→maxy*);
        }
      }
      break;
    case *mp_path_type*: *mp_path_bbox*(*mp*, *cur_exp_knot*( ));
      break;
    case *mp_pen_type*: *mp_pen_bbox*(*mp*, *cur_exp_knot*( ));
      break;
    default: **return** *false*;
    }
    return *true*;
  }

985.   Here is a routine that interprets *cur_exp* as a file name and tries to read a line from the file or to close the file.

⟨Declare unary action procedures 960⟩ +≡

  static void *mp_do_read_or_close*(**MP** *mp*, **quarterword** *c*)

  {

    mp_value *new_expr*;

    readf_index *n*, *n0*;   /∗ indices for searching *rd_fname* ∗/

    memset(&*new_expr*, 0, **sizeof**(**mp_value**));

    new_number(*new_expr.data.n*);   /∗ Find the *n* where *rd_fname*[*n*] = *cur_exp*; if *cur_exp* must be inserted, call *start_read_input* and **goto** *found* or *not_found* ∗/

      /∗ Free slots in the *rd_file* and *rd_fname* arrays are marked with NULL's in *rd_fname*. ∗/

    {

      char ∗*fn*;

      n = *mp*→*read_files*;

      n0 = *mp*→*read_files*;

      fn = *mp_xstrdup*(*mp*, *mp_str*(*mp*, *cur_exp_str*()));

      while (*mp_xstrcmp*(*fn*, *mp*→*rd_fname*[*n*]) ≠ 0) {

        if (*n* > 0) {

          decr(*n*);

        }

        else if (*c* ≡ *mp_close_from_op*) {

          goto CLOSE_FILE;

        }

        else {

          if (*n0* ≡ *mp*→*read_files*) {

            if (*mp*→*read_files* < *mp*→*max_read_files*) {

              incr(*mp*→*read_files*);

            }

            else {

              void ∗∗*rd_file*;

              char ∗∗*rd_fname*;

              readf_index *l*, *k*;

              l = *mp*→*max_read_files* + (*mp*→*max_read_files*/4);

              rd_file = *xmalloc*((*l* + 1), **sizeof**(**void** ∗));

              rd_fname = *xmalloc*((*l* + 1), **sizeof**(**char** ∗));

              for (*k* = 0; *k* ≤ *l*; *k*++) {

                if (*k* ≤ *mp*→*max_read_files*) {

                  rd_file[*k*] = *mp*→*rd_file*[*k*];

                  rd_fname[*k*] = *mp*→*rd_fname*[*k*];

                }

                else {

                  rd_file[*k*] = 0;

                  rd_fname[*k*] = Λ;

                }

              }

              xfree(*mp*→*rd_file*);

              xfree(*mp*→*rd_fname*);

              mp→*max_read_files* = *l*;

              mp→*rd_file* = *rd_file*;

              mp→*rd_fname* = *rd_fname*;

            }

```
        }
      n = n0 ;
      if (mp_start_read_input (mp, fn, n)) goto FOUND;
      else goto NOT_FOUND;
    }
    if (mp→rd_fname [n] ≡ Λ) {
      n0 = n;
    }
  }
  if (c ≡ mp_close_from_op) {
    (mp→close_file )(mp, mp→rd_file [n]);
    goto NOT_FOUND;
  }
}
mp_begin_file_reading (mp);
name = is_read ;
if (mp_input_ln (mp, mp→rd_file [n])) goto FOUND;
mp_end_file_reading (mp);
NOT_FOUND:      /∗ Record the end of file and set cur_exp to a dummy value ∗/
xfree (mp→rd_fname [n]);
mp→rd_fname [n] = Λ;
if (n ≡ mp→read_files − 1)  mp→read_files = n;
if (c ≡ mp_close_from_op) goto CLOSE_FILE;
new_expr .data .str = mp→eof_line ;
add_str_ref (new_expr .data .str );
mp_flush_cur_exp (mp, new_expr );
mp→cur_exp.type = mp_string_type ;
return ;
CLOSE_FILE: mp_flush_cur_exp (mp, new_expr );
mp→cur_exp.type = mp_vacuous ;
return ;
FOUND: mp_flush_cur_exp (mp, new_expr );
mp_finish_read (mp);
}
```

986. The string denoting end-of-file is a one-byte string at position zero, by definition. I have to cheat a little here because

⟨ Global variables 14 ⟩ +≡
  mp_string *eof_line* ;

987.   ⟨ Set initial values of key variables 38 ⟩ +≡
  $mp→eof\_line = mp\_rtsl (mp, "\backslash 0", 1);$
  $mp→eof\_line→refs = \texttt{MAX\_STR\_REF};$

988.   Finally, we have the operations that combine a capsule p with the current expression.

Several of the binary operations are potentially complicated by the fact that *independent* values can sneak into capsules. For example, we've seen an instance of this difficulty in the unary operation of negation. In order to reduce the number of cases that need to be handled, we first change the two operands (if necessary) to rid them of *independent* components. The original operands are put into capsules called *old_p* and *old_exp*, which will be recycled after the binary operation has been safely carried out.

#define *binary_return*
>   {
>   *mp_finish_binary*(*mp*, *old_p*, *old_exp*);
>   **return**;
>   }

⟨ Declare binary action procedures 989 ⟩;

static void *mp_finish_binary*(**MP** *mp*, **mp_node** *old_p*, **mp_node** *old_exp*)
{
    check_arith( );   /∗ Recycle any sidestepped *independent* capsules ∗/
    if (*old_p* ≠ Λ) {
        mp_recycle_value(*mp*, *old_p*);
        mp_free_value_node(*mp*, *old_p*);
    }
    if (*old_exp* ≠ Λ) {
        mp_recycle_value(*mp*, *old_exp*);
        mp_free_value_node(*mp*, *old_exp*);
    }
}

static void *mp_do_binary*(**MP** *mp*, **mp_node** *p*, **integer** *c*)
{
    mp_node *q*, *r*, *rr*;   /∗ for list manipulation ∗/
    mp_node *old_p*, *old_exp*;   /∗ capsules to recycle ∗/
    mp_value *new_expr*;

    check_arith( );
    if (*number_greater*(*internal_value*(*mp_tracing_commands*), *two_t*)) {
        /∗ Trace the current binary operation ∗/
        mp_begin_diagnostic(*mp*);
        mp_print_nl(*mp*, "{(");
        mp_print_exp(*mp*, *p*, 0);   /∗ show the operand, but not verbosely ∗/
        mp_print_char(*mp*, *xord*(')'));
        mp_print_op(*mp*, (**quarterword**) *c*);
        mp_print_char(*mp*, *xord*('('));
        mp_print_exp(*mp*, Λ, 0);
        mp_print(*mp*, ")}");
        mp_end_diagnostic(*mp*, *false*);
    }   /∗ Sidestep *independent* cases in capsule *p* ∗/   /∗ A big node is considered to be "tarnished"
            if it contains at least one independent component. We will define a simple function called
            '*tarnished*' that returns Λ if and only if its argument is not tarnished. ∗/
    switch (*mp_type*(*p*)) {
    case *mp_transform_type*: **case** *mp_color_type*: **case** *mp_cmykcolor_type*: **case** *mp_pair_type*:
        old_p = *mp_tarnished*(*mp*, *p*);
        break;
    case *mp_independent*: *old_p* = MP_VOID;
        break;
    default: *old_p* = Λ;

```
    break;
  }
  if (old_p ≠ Λ) {
    q = mp_stash_cur_exp(mp);
    old_p = p;
    mp_make_exp_copy(mp, old_p);
    p = mp_stash_cur_exp(mp);
    mp_unstash_cur_exp(mp, q);
  }    /* Sidestep independent cases in the current expression */
  switch (mp→cur_exp.type) {
  case mp_transform_type: case mp_color_type: case mp_cmykcolor_type: case mp_pair_type:
    old_exp = mp_tarnished(mp, cur_exp_node());
    break;
  case mp_independent: old_exp = MP_VOID;
    break;
  default: old_exp = Λ;
    break;
  }
  if (old_exp ≠ Λ) {
    old_exp = cur_exp_node();
    mp_make_exp_copy(mp, old_exp);
  }
  switch (c) {
  case mp_plus: case mp_minus:       /* Add or subtract the current expression from p */
    if ((mp→cur_exp.type < mp_color_type) ∨ (mp_type(p) < mp_color_type)) {
      mp_bad_binary(mp, p, (quarterword) c);
    }
    else {
      quarterword cc = (quarterword) c;
      if ((mp→cur_exp.type > mp_pair_type) ∧ (mp_type(p) > mp_pair_type)) {
        mp_add_or_subtract(mp, p, Λ, cc);
      }
      else {
        if (mp→cur_exp.type ≠ mp_type(p)) {
          mp_bad_binary(mp, p, cc);
        }
        else {
          q = value_node(p);
          r = value_node(cur_exp_node());
          switch (mp→cur_exp.type) {
          case mp_pair_type: mp_add_or_subtract(mp, x_part(q), x_part(r), cc);
            mp_add_or_subtract(mp, y_part(q), y_part(r), cc);
            break;
          case mp_color_type: mp_add_or_subtract(mp, red_part(q), red_part(r), cc);
            mp_add_or_subtract(mp, green_part(q), green_part(r), cc);
            mp_add_or_subtract(mp, blue_part(q), blue_part(r), cc);
            break;
          case mp_cmykcolor_type: mp_add_or_subtract(mp, cyan_part(q), cyan_part(r), cc);
            mp_add_or_subtract(mp, magenta_part(q), magenta_part(r), cc);
            mp_add_or_subtract(mp, yellow_part(q), yellow_part(r), cc);
            mp_add_or_subtract(mp, black_part(q), black_part(r), cc);
            break;
```

    case $mp\_transform\_type$: $mp\_add\_or\_subtract(mp, tx\_part(q), tx\_part(r), cc)$;
     $mp\_add\_or\_subtract(mp, ty\_part(q), ty\_part(r), cc)$;
     $mp\_add\_or\_subtract(mp, xx\_part(q), xx\_part(r), cc)$;
     $mp\_add\_or\_subtract(mp, xy\_part(q), xy\_part(r), cc)$;
     $mp\_add\_or\_subtract(mp, yx\_part(q), yx\_part(r), cc)$;
     $mp\_add\_or\_subtract(mp, yy\_part(q), yy\_part(r), cc)$;
     break;
    default: /* there are no other valid cases, but please the compiler */
     break;
   }
  }
 }
}
break;
case mp_less_than: **case** $mp\_less\_or\_equal$: **case** $mp\_greater\_than$: **case** $mp\_greater\_or\_equal$:
 case mp_equal_to: **case** $mp\_unequal\_to$: $check\_arith(\,)$;
  /* at this point $arith\_error$ should be $false$? */
 if $((mp \rightarrow cur\_exp.type > mp\_pair\_type) \wedge (mp\_type(p) > mp\_pair\_type))$ {
  $mp\_add\_or\_subtract(mp, p, \Lambda, mp\_minus)$; /* $cur\_exp: = (p) - cur\_exp$ */
 }
 else if $(mp \rightarrow cur\_exp.type \neq mp\_type(p))$ {
  $mp\_bad\_binary(mp, p, (\textbf{quarterword})\ c)$;
  goto DONE;
 }
 else if $(mp \rightarrow cur\_exp.type \equiv mp\_string\_type)$ {
  $memset(\&new\_expr, 0, \textbf{sizeof}(\textbf{mp\_value}))$;
  $new\_number(new\_expr.data.n)$;
  $set\_number\_from\_scaled(new\_expr.data.n, mp\_str\_vs\_str(mp, value\_str(p), cur\_exp\_str(\,)))$;
  $mp\_flush\_cur\_exp(mp, new\_expr)$;
 }
 else if $((mp \rightarrow cur\_exp.type \equiv mp\_unknown\_string) \vee (mp \rightarrow cur\_exp.type \equiv mp\_unknown\_boolean))$ {
   /* Check if unknowns have been equated */
   /* When two unknown strings are in the same ring, we know that they are equal. Otherwise,
    we don't know whether they are equal or not, so we make no change. */
  $q = value\_node(cur\_exp\_node(\,))$;
  while $((q \neq cur\_exp\_node(\,)) \wedge (q \neq p))$ $q = value\_node(q)$;
  if $(q \equiv p)$ {
   $memset(\&new\_expr, 0, \textbf{sizeof}(\textbf{mp\_value}))$;
   $new\_number(new\_expr.data.n)$;
   $set\_cur\_exp\_node(\Lambda)$;
   $mp\_flush\_cur\_exp(mp, new\_expr)$;
  }
 }
 else if $((mp \rightarrow cur\_exp.type \leq mp\_pair\_type) \wedge (mp \rightarrow cur\_exp.type \geq mp\_transform\_type))$ {
   /* Reduce comparison of big nodes to comparison of scalars */ /* In the following, the
    while loops exist just so that **break** can be used, each loop runs exactly once. */
  quarterword $part\_type$;

  $q = value\_node(p)$;
  $r = value\_node(cur\_exp\_node(\,))$;
  $part\_type = 0$;
  switch $(mp \rightarrow cur\_exp.type)$ {
  case mp_pair_type:

```
      while (part_type ≡ 0) {
        rr = x_part(r);
        part_type = mp_x_part;
        mp_add_or_subtract(mp, x_part(q), rr, mp_minus);
        if (mp_type(rr) ≠ mp_known ∨ ¬number_zero(value_number(rr))) break;
        rr = y_part(r);
        part_type = mp_y_part;
        mp_add_or_subtract(mp, y_part(q), rr, mp_minus);
        if (mp_type(rr) ≠ mp_known ∨ ¬number_zero(value_number(rr))) break;
      }
      mp_take_part(mp, part_type);
      break;
    case mp_color_type:
      while (part_type ≡ 0) {
        rr = red_part(r);
        part_type = mp_red_part;
        mp_add_or_subtract(mp, red_part(q), rr, mp_minus);
        if (mp_type(rr) ≠ mp_known ∨ ¬number_zero(value_number(rr))) break;
        rr = green_part(r);
        part_type = mp_green_part;
        mp_add_or_subtract(mp, green_part(q), rr, mp_minus);
        if (mp_type(rr) ≠ mp_known ∨ ¬number_zero(value_number(rr))) break;
        rr = blue_part(r);
        part_type = mp_blue_part;
        mp_add_or_subtract(mp, blue_part(q), rr, mp_minus);
        if (mp_type(rr) ≠ mp_known ∨ ¬number_zero(value_number(rr))) break;
      }
      mp_take_part(mp, part_type);
      break;
    case mp_cmykcolor_type:
      while (part_type ≡ 0) {
        rr = cyan_part(r);
        part_type = mp_cyan_part;
        mp_add_or_subtract(mp, cyan_part(q), rr, mp_minus);
        if (mp_type(rr) ≠ mp_known ∨ ¬number_zero(value_number(rr))) break;
        rr = magenta_part(r);
        part_type = mp_magenta_part;
        mp_add_or_subtract(mp, magenta_part(q), rr, mp_minus);
        if (mp_type(rr) ≠ mp_known ∨ ¬number_zero(value_number(rr))) break;
        rr = yellow_part(r);
        part_type = mp_yellow_part;
        mp_add_or_subtract(mp, yellow_part(q), rr, mp_minus);
        if (mp_type(rr) ≠ mp_known ∨ ¬number_zero(value_number(rr))) break;
        rr = black_part(r);
        part_type = mp_black_part;
        mp_add_or_subtract(mp, black_part(q), rr, mp_minus);
        if (mp_type(rr) ≠ mp_known ∨ ¬number_zero(value_number(rr))) break;
      }
      mp_take_part(mp, part_type);
      break;
    case mp_transform_type:
      while (part_type ≡ 0) {
```

$rr = tx\_part(r);$
$part\_type = mp\_x\_part;$
$mp\_add\_or\_subtract(mp, tx\_part(q), rr, mp\_minus);$
if $(mp\_type(rr) \neq mp\_known \lor \neg number\_zero(value\_number(rr)))$ **break**;
$rr = ty\_part(r);$
$part\_type = mp\_y\_part;$
$mp\_add\_or\_subtract(mp, ty\_part(q), rr, mp\_minus);$
if $(mp\_type(rr) \neq mp\_known \lor \neg number\_zero(value\_number(rr)))$ **break**;
$rr = xx\_part(r);$
$part\_type = mp\_xx\_part;$
$mp\_add\_or\_subtract(mp, xx\_part(q), rr, mp\_minus);$
if $(mp\_type(rr) \neq mp\_known \lor \neg number\_zero(value\_number(rr)))$ **break**;
$rr = xy\_part(r);$
$part\_type = mp\_xy\_part;$
$mp\_add\_or\_subtract(mp, xy\_part(q), rr, mp\_minus);$
if $(mp\_type(rr) \neq mp\_known \lor \neg number\_zero(value\_number(rr)))$ **break**;
$rr = yx\_part(r);$
$part\_type = mp\_yx\_part;$
$mp\_add\_or\_subtract(mp, yx\_part(q), rr, mp\_minus);$
if $(mp\_type(rr) \neq mp\_known \lor \neg number\_zero(value\_number(rr)))$ **break**;
$rr = yy\_part(r);$
$part\_type = mp\_yy\_part;$
$mp\_add\_or\_subtract(mp, yy\_part(q), rr, mp\_minus);$
if $(mp\_type(rr) \neq mp\_known \lor \neg number\_zero(value\_number(rr)))$ **break**;
$\}$
$mp\_take\_part(mp, part\_type);$
break;
default: $assert(0);$ /* todo: $mp{\rightarrow}cur\_exp.type > mp\_transform\_node\_type$? */
break;
$\}$
$\}$
else if $(mp{\rightarrow}cur\_exp.type \equiv mp\_boolean\_type)$ {
$memset(\&new\_expr, 0, \textbf{sizeof}(\textbf{mp\_value}));$
$new\_number(new\_expr.data.n);$
$set\_number\_from\_boolean(new\_expr.data.n,$
$\quad number\_to\_scaled(cur\_exp\_value\_number()) - number\_to\_scaled(value\_number(p)));$
$mp\_flush\_cur\_exp(mp, new\_expr);$
$\}$
else {
$mp\_bad\_binary(mp, p, (\textbf{quarterword})\ c);$
goto DONE;
$\}$ /* Compare the current expression with zero */
if $(mp{\rightarrow}cur\_exp.type \neq mp\_known)$ {
const char $*hlp[\ ] = \{$"Oh␣dear.␣␣I␣can\'t␣decide␣if␣the␣expression␣above␣is␣positive,",
"negative,␣or␣zero.␣␣So␣this␣comparison␣test␣won't␣be␣'true'.", $\Lambda\};$
if $(mp{\rightarrow}cur\_exp.type < mp\_known)$ {
$mp\_disp\_err(mp, p);$
$hlp[0] = $"The␣quantities␣shown␣above␣have␣not␣been␣equated.";
$hlp[1] = \Lambda;$
$\}$
$mp\_disp\_err(mp, \Lambda);$
$memset(\&new\_expr, 0, \textbf{sizeof}(\textbf{mp\_value}));$

```
      new_number(new_expr.data.n);
      set_number_from_boolean(new_expr.data.n, mp_false_code);
      mp_back_error(mp, "Unknown␣relation␣will␣be␣considered␣false", hlp, true);
      ;
      mp_get_x_next(mp);
      mp_flush_cur_exp(mp, new_expr);
    }
    else {
      switch (c) {
      case mp_less_than: boolean_reset(number_negative(cur_exp_value_number( )));
        break;
      case mp_less_or_equal: boolean_reset(number_nonpositive(cur_exp_value_number( )));
        break;
      case mp_greater_than: boolean_reset(number_positive(cur_exp_value_number( )));
        break;
      case mp_greater_or_equal: boolean_reset(number_nonnegative(cur_exp_value_number( )));
        break;
      case mp_equal_to: boolean_reset(number_zero(cur_exp_value_number( )));
        break;
      case mp_unequal_to: boolean_reset(number_nonzero(cur_exp_value_number( )));
        break;
      }
      ;     /* there are no other cases */
    }
    mp→cur_exp.type = mp_boolean_type;
DONE: mp→arith_error = false;       /* ignore overflow in comparisons */
    break;
  case mp_and_op: case mp_or_op:
      /* Here we use the sneaky fact that and_op − false_code = or_op − true_code */
    if ((mp_type(p) ≠ mp_boolean_type) ∨ (mp→cur_exp.type ≠ mp_boolean_type))
      mp_bad_binary(mp, p, (quarterword) c);
    else if (number_to_boolean(p→data.n) ≡ c + mp_false_code − mp_and_op) {
      set_cur_exp_value_boolean(number_to_boolean(p→data.n));
    }
    break;
  case mp_times:
    if ((mp→cur_exp.type < mp_color_type) ∨ (mp_type(p) < mp_color_type)) {
      mp_bad_binary(mp, p, mp_times);
    }
    else if ((mp→cur_exp.type ≡ mp_known) ∨ (mp_type(p) ≡ mp_known)) {
        /* Multiply when at least one operand is known */
      mp_number vv;

      new_fraction(vv);
      if (mp_type(p) ≡ mp_known) {
        number_clone(vv, value_number(p));
        mp_free_value_node(mp, p);
      }
      else {
        number_clone(vv, cur_exp_value_number( ));
        mp_unstash_cur_exp(mp, p);
      }
      if (mp→cur_exp.type ≡ mp_known) {
```

```
      mp_number ret;
      new_number(ret);
      take_scaled(ret, cur_exp_value_number( ), vv);
      set_cur_exp_value_number(ret);
      free_number(ret);
    }
    else if (mp→cur_exp.type ≡ mp_pair_type) {
      mp_dep_mult(mp, (mp_value_node) x_part(value_node(cur_exp_node( ))), vv, true);
      mp_dep_mult(mp, (mp_value_node) y_part(value_node(cur_exp_node( ))), vv, true);
    }
    else if (mp→cur_exp.type ≡ mp_color_type) {
      mp_dep_mult(mp, (mp_value_node) red_part(value_node(cur_exp_node( ))), vv, true);
      mp_dep_mult(mp, (mp_value_node) green_part(value_node(cur_exp_node( ))), vv, true);
      mp_dep_mult(mp, (mp_value_node) blue_part(value_node(cur_exp_node( ))), vv, true);
    }
    else if (mp→cur_exp.type ≡ mp_cmykcolor_type) {
      mp_dep_mult(mp, (mp_value_node) cyan_part(value_node(cur_exp_node( ))), vv, true);
      mp_dep_mult(mp, (mp_value_node) magenta_part(value_node(cur_exp_node( ))), vv, true);
      mp_dep_mult(mp, (mp_value_node) yellow_part(value_node(cur_exp_node( ))), vv, true);
      mp_dep_mult(mp, (mp_value_node) black_part(value_node(cur_exp_node( ))), vv, true);
    }
    else {
      mp_dep_mult(mp, Λ, vv, true);
    }
    free_number(vv);
    binary_return;
  }
  else if ((mp_nice_color_or_pair(mp, p,
        mp_type(p)) ∧ (mp→cur_exp.type > mp_pair_type)) ∨ (mp_nice_color_or_pair(mp,
        cur_exp_node( ), mp→cur_exp.type) ∧ (mp_type(p) > mp_pair_type))) {
    mp_hard_times(mp, p);
    binary_return;
  }
  else {
    mp_bad_binary(mp, p, mp_times);
  }
  break;
case mp_over:
  if ((mp→cur_exp.type ≠ mp_known) ∨ (mp_type(p) < mp_color_type)) {
    mp_bad_binary(mp, p, mp_over);
  }
  else {
    mp_number v_n;
    new_number(v_n);
    number_clone(v_n, cur_exp_value_number( ));
    mp_unstash_cur_exp(mp, p);
    if (number_zero(v_n)) {      /* Squeal about division by zero */
      const char *hlp[] = {"You're␣trying␣to␣divide␣the␣quantity␣shown␣above␣the␣error",
          "message␣by␣zero.␣I'm␣going␣to␣divide␣it␣by␣one␣instead.", Λ};
      mp_disp_err(mp, Λ);
      mp_back_error(mp, "Division␣by␣zero", hlp, true);
```

```
            mp_get_x_next(mp);
        }
        else {
            if (mp→cur_exp.type ≡ mp_known) {
                mp_number ret;

                new_number(ret);
                make_scaled(ret, cur_exp_value_number( ), v_n);
                set_cur_exp_value_number(ret);
                free_number(ret);
            }
            else if (mp→cur_exp.type ≡ mp_pair_type) {
                mp_dep_div(mp, (mp_value_node) x_part(value_node(cur_exp_node( ))), v_n);
                mp_dep_div(mp, (mp_value_node) y_part(value_node(cur_exp_node( ))), v_n);
            }
            else if (mp→cur_exp.type ≡ mp_color_type) {
                mp_dep_div(mp, (mp_value_node) red_part(value_node(cur_exp_node( ))), v_n);
                mp_dep_div(mp, (mp_value_node) green_part(value_node(cur_exp_node( ))), v_n);
                mp_dep_div(mp, (mp_value_node) blue_part(value_node(cur_exp_node( ))), v_n);
            }
            else if (mp→cur_exp.type ≡ mp_cmykcolor_type) {
                mp_dep_div(mp, (mp_value_node) cyan_part(value_node(cur_exp_node( ))), v_n);
                mp_dep_div(mp, (mp_value_node) magenta_part(value_node(cur_exp_node( ))), v_n);
                mp_dep_div(mp, (mp_value_node) yellow_part(value_node(cur_exp_node( ))), v_n);
                mp_dep_div(mp, (mp_value_node) black_part(value_node(cur_exp_node( ))), v_n);
            }
            else {
                mp_dep_div(mp, Λ, v_n);
            }
        }
        free_number(v_n);
        binary_return;
    }
    break;
case mp_pythag_add: case mp_pythag_sub:
    if ((mp→cur_exp.type ≡ mp_known) ∧ (mp_type(p) ≡ mp_known)) {
        mp_number r;

        new_number(r);
        if (c ≡ mp_pythag_add) {
            pyth_add(r, value_number(p), cur_exp_value_number( ));
        }
        else {
            pyth_sub(r, value_number(p), cur_exp_value_number( ));
        }
        set_cur_exp_value_number(r);
        free_number(r);
    }
    else  mp_bad_binary(mp, p, (quarterword) c);
    break;
case mp_rotated_by: case mp_slanted_by: case mp_scaled_by: case mp_shifted_by:
    case mp_transformed_by: case mp_x_scaled: case mp_y_scaled: case mp_z_scaled:
        /* The next few sections of the program deal with affine transformations of coordinate data. */
        if (mp_type(p) ≡ mp_path_type) {
```

```
      path_trans((quarterword) c, p);
      binary_return;
    }
  else if (mp_type(p) ≡ mp_pen_type) {
    pen_trans((quarterword) c, p);
    set_cur_exp_knot(mp_convex_hull(mp, cur_exp_knot( )));
        /* rounding error could destroy convexity */
    binary_return;
  }
  else if ((mp_type(p) ≡ mp_pair_type) ∨ (mp_type(p) ≡ mp_transform_type)) {
    mp_big_trans(mp, p, (quarterword) c);
  }
  else if (mp_type(p) ≡ mp_picture_type) {
    mp_do_edges_trans(mp, p, (quarterword) c);
    binary_return;
  }
  else {
    mp_bad_binary(mp, p, (quarterword) c);
  }
  break;
case mp_concatenate:
  if ((mp→cur_exp.type ≡ mp_string_type) ∧ (mp_type(p) ≡ mp_string_type)) {
    mp_string str = mp_cat(mp, value_str(p), cur_exp_str( ));

    delete_str_ref(cur_exp_str( ));
    set_cur_exp_str(str);
  }
  else  mp_bad_binary(mp, p, mp_concatenate);
  break;
case mp_substring_of:
  if (mp_nice_pair(mp, p, mp_type(p)) ∧ (mp→cur_exp.type ≡ mp_string_type)) {
    mp_string str = mp_chop_string(mp, cur_exp_str( ),
        round_unscaled(value_number(x_part(value_node(p)))),
        round_unscaled(value_number(y_part(value_node(p)))));

    delete_str_ref(cur_exp_str( ));
    set_cur_exp_str(str);
  }
  else  mp_bad_binary(mp, p, mp_substring_of);
  break;
case mp_subpath_of:
  if (mp→cur_exp.type ≡ mp_pair_type)  mp_pair_to_path(mp);
  if (mp_nice_pair(mp, p, mp_type(p)) ∧ (mp→cur_exp.type ≡ mp_path_type))
    mp_chop_path(mp, value_node(p));
  else  mp_bad_binary(mp, p, mp_subpath_of);
  break;
case mp_point_of: case mp_precontrol_of: case mp_postcontrol_of:
  if (mp→cur_exp.type ≡ mp_pair_type)  mp_pair_to_path(mp);
  if ((mp→cur_exp.type ≡ mp_path_type) ∧ (mp_type(p) ≡ mp_known))
    mp_find_point(mp, value_number(p), (quarterword) c);
  else  mp_bad_binary(mp, p, (quarterword) c);
  break;
case mp_pen_offset_of:
```

if $((mp \rightarrow cur\_exp.type \equiv mp\_pen\_type) \wedge mp\_nice\_pair(mp, p, mp\_type(p)))$
   $mp\_set\_up\_offset(mp, value\_node(p))$;
else $mp\_bad\_binary(mp, p, mp\_pen\_offset\_of)$;
break;
case $mp\_direction\_time\_of$:
  if $(mp \rightarrow cur\_exp.type \equiv mp\_pair\_type)$ $mp\_pair\_to\_path(mp)$;
  if $((mp \rightarrow cur\_exp.type \equiv mp\_path\_type) \wedge mp\_nice\_pair(mp, p, mp\_type(p)))$
   $mp\_set\_up\_direction\_time(mp, value\_node(p))$;
  else $mp\_bad\_binary(mp, p, mp\_direction\_time\_of)$;
  break;
case $mp\_envelope\_of$:
  if $((mp\_type(p) \neq mp\_pen\_type) \vee (mp \rightarrow cur\_exp.type \neq mp\_path\_type))$
   $mp\_bad\_binary(mp, p, mp\_envelope\_of)$;
  else $mp\_set\_up\_envelope(mp, p)$;
  break;
case mp_glyph_infont:
  if $((mp\_type(p) \neq mp\_string\_type \wedge mp\_type(p) \neq mp\_known) \vee (mp \rightarrow cur\_exp.type \neq mp\_string\_type))$
   $mp\_bad\_binary(mp, p, mp\_glyph\_infont)$;
  else $mp\_set\_up\_glyph\_infont(mp, p)$;
  break;
case $mp\_arc\_time\_of$:
  if $(mp \rightarrow cur\_exp.type \equiv mp\_pair\_type)$ $mp\_pair\_to\_path(mp)$;
  if $((mp \rightarrow cur\_exp.type \equiv mp\_path\_type) \wedge (mp\_type(p) \equiv mp\_known))$ {
   $memset(\&new\_expr, 0, \textbf{sizeof}(\textbf{mp\_value}))$;
   $new\_number(new\_expr.data.n)$;
   $mp\_get\_arc\_time(mp, \&new\_expr.data.n, cur\_exp\_knot(), value\_number(p))$;
   $mp\_flush\_cur\_exp(mp, new\_expr)$;
  }
  else {
   $mp\_bad\_binary(mp, p, (\textbf{quarterword})\ c)$;
  }
  break;
case $mp\_intersect$:
  if $(mp\_type(p) \equiv mp\_pair\_type)$ {
   $q = mp\_stash\_cur\_exp(mp)$;
   $mp\_unstash\_cur\_exp(mp, p)$;
   $mp\_pair\_to\_path(mp)$;
   $p = mp\_stash\_cur\_exp(mp)$;
   $mp\_unstash\_cur\_exp(mp, q)$;
  }
  if $(mp \rightarrow cur\_exp.type \equiv mp\_pair\_type)$ $mp\_pair\_to\_path(mp)$;
  if $((mp \rightarrow cur\_exp.type \equiv mp\_path\_type) \wedge (mp\_type(p) \equiv mp\_path\_type))$ {
   mp_number $arg1, arg2$;

   $new\_number(arg1)$;
   $new\_number(arg2)$;
   $mp\_path\_intersection(mp, value\_knot(p), cur\_exp\_knot())$;
   $number\_clone(arg1, mp \rightarrow cur\_t)$;
   $number\_clone(arg2, mp \rightarrow cur\_tt)$;
   $mp\_pair\_value(mp, arg1, arg2)$;
   $free\_number(arg1)$;
   $free\_number(arg2)$;
  }

```
    else {
      mp_bad_binary(mp, p, mp_intersect);
    }
    break;
  case mp_in_font:
    if ((mp⁃cur_exp.type ≠ mp_string_type) ∨ mp_type(p) ≠ mp_string_type) {
      mp_bad_binary(mp, p, mp_in_font);
    }
    else {
      mp_do_infont(mp, p);
      binary_return;
    }
    break;
  }      /∗ there are no other cases ∗/
  mp_recycle_value(mp, p);
  mp_free_value_node(mp, p);      /∗ return to avoid this ∗/
  mp_finish_binary(mp, old_p, old_exp);
}
```

989. ⟨Declare binary action procedures 989⟩ ≡

```
static void mp_bad_binary(MP mp, mp_node p, quarterword c)
{
    char msg[256];
    mp_string sname;
    int old_setting = mp→selector;
    const char *hlp[] = {"I'm␣afraid␣I␣don't␣know␣how␣to␣apply␣that␣operation␣to␣that",
        "combination␣of␣types.␣Continue,␣and␣I'll␣return␣the␣second",
        "argument␣(see␣above)␣as␣the␣result␣of␣the␣operation.", Λ};

    mp→selector = new_string;
    if (c ≥ mp_min_of) mp_print_op(mp, c);
    mp_print_known_or_unknown_type(mp, mp_type(p), p);
    if (c ≥ mp_min_of) mp_print(mp, "of");
    else  mp_print_op(mp, c);
    mp_print_known_or_unknown_type(mp, mp→cur_exp.type, cur_exp_node());
    sname = mp_make_string(mp);
    mp→selector = old_setting;
    mp_snprintf(msg, 256, "Not␣implemented:␣%s", mp_str(mp, sname));
    ;
    delete_str_ref(sname);
    mp_disp_err(mp, p);
    mp_disp_err(mp, Λ);
    mp_back_error(mp, msg, hlp, true);
    mp_get_x_next(mp);
}

static void mp_bad_envelope_pen(MP mp)
{
    const char *hlp[] = {"I'm␣afraid␣I␣don't␣know␣how␣to␣apply␣that␣operation␣to␣that",
        "combination␣of␣types.␣Continue,␣and␣I'll␣return␣the␣second",
        "argument␣(see␣above)␣as␣the␣result␣of␣the␣operation.", Λ};

    mp_disp_err(mp, Λ);
    mp_disp_err(mp, Λ);
    mp_back_error(mp, "Not␣implemented:␣envelope(elliptical␣pen)of(path)", hlp, true);
    ;
    mp_get_x_next(mp);
}
```

See also sections 990, 991, 993, 996, 997, 998, 1005, 1006, 1007, 1008, 1009, 1019, 1027, 1028, 1029, 1030, and 1031.

This code is used in section 988.

990.   ⟨Declare binary action procedures 989⟩ +≡
  static mp_node *mp_tarnished*(**MP** *mp*, **mp_node** *p*)
  {
    mp_node *q*;   /* beginning of the big node */
    mp_node *r*;   /* moving value node pointer */

    (**void**) *mp*;
    q = *value_node*(*p*);
    switch (*mp_type*(*p*)) {
    case *mp_pair_type*: *r* = *x_part*(*q*);
      if (*mp_type*(*r*) ≡ *mp_independent*) **return** MP_VOID;
      r = *y_part*(*q*);
      if (*mp_type*(*r*) ≡ *mp_independent*) **return** MP_VOID;
      break;
    case *mp_color_type*: *r* = *red_part*(*q*);
      if (*mp_type*(*r*) ≡ *mp_independent*) **return** MP_VOID;
      r = *green_part*(*q*);
      if (*mp_type*(*r*) ≡ *mp_independent*) **return** MP_VOID;
      r = *blue_part*(*q*);
      if (*mp_type*(*r*) ≡ *mp_independent*) **return** MP_VOID;
      break;
    case *mp_cmykcolor_type*: *r* = *cyan_part*(*q*);
      if (*mp_type*(*r*) ≡ *mp_independent*) **return** MP_VOID;
      r = *magenta_part*(*q*);
      if (*mp_type*(*r*) ≡ *mp_independent*) **return** MP_VOID;
      r = *yellow_part*(*q*);
      if (*mp_type*(*r*) ≡ *mp_independent*) **return** MP_VOID;
      r = *black_part*(*q*);
      if (*mp_type*(*r*) ≡ *mp_independent*) **return** MP_VOID;
      break;
    case *mp_transform_type*: *r* = *tx_part*(*q*);
      if (*mp_type*(*r*) ≡ *mp_independent*) **return** MP_VOID;
      r = *ty_part*(*q*);
      if (*mp_type*(*r*) ≡ *mp_independent*) **return** MP_VOID;
      r = *xx_part*(*q*);
      if (*mp_type*(*r*) ≡ *mp_independent*) **return** MP_VOID;
      r = *xy_part*(*q*);
      if (*mp_type*(*r*) ≡ *mp_independent*) **return** MP_VOID;
      r = *yx_part*(*q*);
      if (*mp_type*(*r*) ≡ *mp_independent*) **return** MP_VOID;
      r = *yy_part*(*q*);
      if (*mp_type*(*r*) ≡ *mp_independent*) **return** MP_VOID;
      break;
    default: /* there are no other valid cases, but please the compiler */
      break;
    }
    return Λ;
  }

991.   The first argument to *add_or_subtract* is the location of a value node in a capsule or pair node that will soon be recycled. The second argument is either a location within a pair or transform node of *cur_exp*, or it is NULL (which means that *cur_exp* itself should be the second argument). The third argument is either *plus* or *minus*.

The sum or difference of the numeric quantities will replace the second operand. Arithmetic overflow may go undetected; users aren't supposed to be monkeying around with really big values.

⟨ Declare binary action procedures 989 ⟩ +≡
  ⟨ Declare the procedure called *dep_finish* 992 ⟩;

  static void *mp_add_or_subtract*(**MP** *mp*, **mp_node** *p*, **mp_node** *q*, **quarterword** *c*)
  {
    mp_variable_types, *t*;   /* operand types */
    mp_value_node *r*;   /* dependency list traverser */
    mp_value_node *v* = Λ;   /* second operand value for dep lists */
    mp_number *vv*;   /* second operand value for known values */
    new_number(*vv*);
    if (*q* ≡ Λ) {
      t = *mp→cur_exp.type*;
      if (*t* < *mp_dependent*) *number_clone*(*vv*, *cur_exp_value_number*( ));
      else *v* = (**mp_value_node**) *dep_list*((**mp_value_node**) *cur_exp_node*( ));
    }
    else {
      t = *mp_type*(*q*);
      if (*t* < *mp_dependent*) *number_clone*(*vv*, *value_number*(*q*));
      else *v* = (**mp_value_node**) *dep_list*((**mp_value_node**) *q*);
    }
    if (*t* ≡ *mp_known*) {
      mp_value_node *qq* = (**mp_value_node**) *q*;

      if (*c* ≡ *mp_minus*) *number_negate*(*vv*);
      if (*mp_type*(*p*) ≡ *mp_known*) {
        slow_add(*vv*, *value_number*(*p*), *vv*);
        if (*q* ≡ Λ) *set_cur_exp_value_number*(*vv*);
        else *set_value_number*(*q*, *vv*);
        free_number(*vv*);
        return;
      }   /* Add a known value to the constant term of *dep_list*(*p*) */
      r = (**mp_value_node**) *dep_list*((**mp_value_node**) *p*);
      while (*dep_info*(*r*) ≠ Λ) *r* = (**mp_value_node**) *mp_link*(*r*);
      slow_add(*vv*, *dep_value*(*r*), *vv*);
      set_dep_value(*r*, *vv*);
      if (*qq* ≡ Λ) {
        qq = *mp_get_dep_node*(*mp*);
        set_cur_exp_node((**mp_node**) *qq*);
        mp→cur_exp.type = *mp_type*(*p*);
        mp_name_type(*qq*) = *mp_capsule*;   /* clang: never read: *q* = (**mp_node**) *qq*; */
      }
      set_dep_list(*qq*, *dep_list*((**mp_value_node**) *p*));
      mp_type(*qq*) = *mp_type*(*p*);
      set_prev_dep(*qq*, *prev_dep*((**mp_value_node**) *p*));
      mp_link(*prev_dep*((**mp_value_node**) *p*)) = (**mp_node**) *qq*;
      mp_type(*p*) = *mp_known*;   /* this will keep the recycler from collecting non-garbage */
    }

```
else {
  if (c ≡ mp_minus) mp_negate_dep_list(mp, v);      /∗ Add operand p to the dependency list v ∗/
      /∗ We prefer dependent lists to mp_proto_dependent ones, because it is nice to retain the extra
        accuracy of fraction coefficients. But we have to handle both kinds, and mixtures too. ∗/
  if (mp_type(p) ≡ mp_known) {       /∗ Add the known value(p) to the constant term of v ∗/
    while (dep_info(v) ≠ Λ) {
      v = (mp_value_node) mp_link(v);
    }
    slow_add(vv, value_number(p), dep_value(v));
    set_dep_value(v, vv);
  }
  else {
    s = mp_type(p);
    r = (mp_value_node) dep_list((mp_value_node) p);
    if (t ≡ mp_dependent) {
      if (s ≡ mp_dependent) {
        mp_number ret1, ret2;

        new_fraction(ret1);
        new_fraction(ret2);
        mp_max_coef(mp, &ret1, r);
        mp_max_coef(mp, &ret2, v);
        number_add(ret1, ret2);
        free_number(ret2);
        if (number_less(ret1, coef_bound_k)) {
          v = mp_p_plus_q(mp, v, r, mp_dependent);
          free_number(ret1);
          goto DONE;
        }
        free_number(ret1);
      }      /∗ fix_needed will necessarily be false ∗/
      t = mp_proto_dependent;
      v = mp_p_over_v(mp, v, unity_t, mp_dependent, mp_proto_dependent);
    }
    if (s ≡ mp_proto_dependent) v = mp_p_plus_q(mp, v, r, mp_proto_dependent);
    else v = mp_p_plus_fq(mp, v, unity_t, r, mp_proto_dependent, mp_dependent);
  DONE:    /∗ Output the answer, v (which might have become known) ∗/
    if (q ≠ Λ) {
      mp_dep_finish(mp, v, (mp_value_node) q, t);
    }
    else {
      mp→cur_exp.type = t;
      mp_dep_finish(mp, v, Λ, t);
    }
  }
}
free_number(vv);
}
```

992.   Here's the current situation: The dependency list v of type t should either be put into the current expression (if $q = \Lambda$) or into location q within a pair node (otherwise). The destination (*cur_exp* or q) formerly held a dependency list with the same final pointer as the list v.

⟨ Declare the procedure called *dep_finish* 992 ⟩ ≡
  static void *mp_dep_finish*(**MP** *mp*, **mp_value_node** *v*, **mp_value_node** *q*, **quarterword** *t*)
  {
    mp_value_node *p*;   /∗ the destination ∗/

    if $(q \equiv \Lambda)$ *p* = (**mp_value_node**) *cur_exp_node*( );
    else *p* = *q*;
    set_dep_list(*p*, *v*);
    mp_type(*p*) = *t*;
    if $(dep\_info(v) \equiv \Lambda)$ {
      mp_number *vv*;   /∗ the value, if it is *known* ∗/

      new_number(*vv*);
      number_clone(*vv*, *value_number*(*v*));
      if $(q \equiv \Lambda)$ {
        mp_value *new_expr*;

        memset(&*new_expr*, 0, **sizeof**(**mp_value**));
        new_number(*new_expr.data.n*);
        number_clone(*new_expr.data.n*, *vv*);
        mp_flush_cur_exp(*mp*, *new_expr*);
      }
      else {
        mp_recycle_value(*mp*, (**mp_node**) *p*);
        mp_type(*q*) = *mp_known*;
        set_value_number(*q*, *vv*);
      }
      free_number(*vv*);
    }
    else if $(q \equiv \Lambda)$ {
      mp→*cur_exp.type* = *t*;
    }
    if $(mp$→*fix_needed*) *mp_fix_dependencies*(*mp*);
  }

This code is used in section 991.

993.   ⟨Declare binary action procedures 989⟩ +≡

  static void mp_dep_mult(**MP** mp, **mp_value_node** p, **mp_number** v, **boolean** v_is_scaled)

  {

    mp_value_node q;   /∗ the dependency list being multiplied by v ∗/

    quarterword s, t;   /∗ its type, before and after ∗/

    if $(p \equiv \Lambda)$ {

      $q = ($**mp_value_node**$)\ cur\_exp\_node($);

    }

    else if $(mp\_type(p) \neq mp\_known)$ {

      $q = p$;

    }

    else {

      {

        mp_number $r1$, $arg1$;

        $new\_number(arg1)$;

        $number\_clone(arg1, dep\_value(p))$;

        if (v_is_scaled) {

          $new\_number(r1)$;

          $take\_scaled(r1, arg1, v)$;

        }

        else {

          $new\_fraction(r1)$;

          $take\_fraction(r1, arg1, v)$;

        }

        $set\_dep\_value(p, r1)$;

        $free\_number(r1)$;

        $free\_number(arg1)$;

      }

      return;

    }

    $t = mp\_type(q)$;

    $q = ($**mp_value_node**$)\ dep\_list(q)$;

    $s = t$;

    if $(t \equiv mp\_dependent)$ {

      if (v_is_scaled) {

        mp_number ab_vs_cd;

        mp_number $arg1$, $arg2$;

        $new\_number(ab\_vs\_cd)$;

        $new\_number(arg2)$;

        $new\_fraction(arg1)$;

        $mp\_max\_coef(mp, \&arg1, q)$;

        $number\_clone(arg2, v)$;

        $number\_abs(arg2)$;

        $ab\_vs\_cd(ab\_vs\_cd, arg1, arg2, coef\_bound\_minus\_1, unity\_t)$;

        $free\_number(arg1)$;

        $free\_number(arg2)$;

        if $(number\_nonnegative(ab\_vs\_cd))$ {

          $t = mp\_proto\_dependent$;

        }

        $free\_number(ab\_vs\_cd)$;

      }

```
  }
  q = mp_p_times_v(mp, q, v, s, t, v_is_scaled);
  mp_dep_finish(mp, q, p, t);
}
```

994.   Here is a routine that is similar to *times*; but it is invoked only internally, when v is a *fraction* whose magnitude is at most 1, and when $cur\_type \geq mp\_color\_type$.

static void $mp\_frac\_mult(\mathbf{MP}\ mp, \mathbf{mp\_number}\ n, \mathbf{mp\_number}\ d)$
$\{$ /∗ multiplies *cur_exp* by n/d ∗/
  mp_node *old_exp*;   /∗ a capsule to recycle ∗/
  mp_number *v*;   /∗ n/d ∗/
  new_fraction(*v*);
  if (*number_greater*(*internal_value*(*mp_tracing_commands*), *two_t*)) $\{$
    ⟨ Trace the fraction multiplication 995 ⟩;
  $\}$
  switch (*mp*→*cur_exp.type*) $\{$
  case *mp_transform_type*: **case** *mp_color_type*: **case** *mp_cmykcolor_type*: **case** *mp_pair_type*:
    old_exp = *mp_tarnished*(*mp*, *cur_exp_node*( ));
    break;
  case *mp_independent*: *old_exp* = MP_VOID;
    break;
  default: *old_exp* = Λ;
    break;
  $\}$
  if (*old_exp* ≠ Λ) $\{$
    old_exp = *cur_exp_node*( );
    mp_make_exp_copy(*mp*, *old_exp*);
  $\}$
  make_fraction(*v*, *n*, *d*);
  if (*mp*→*cur_exp.type* ≡ *mp_known*) $\{$
    mp_number *r1*, *arg1*;

    new_fraction(*r1*);
    new_number(*arg1*);
    number_clone(*arg1*, *cur_exp_value_number*( ));
    take_fraction(*r1*, *arg1*, *v*);
    set_cur_exp_value_number(*r1*);
    free_number(*r1*);
    free_number(*arg1*);
  $\}$
  else if (*mp*→*cur_exp.type* ≡ *mp_pair_type*) $\{$
    mp_dep_mult(*mp*, (**mp_value_node**) *x_part*(*value_node*(*cur_exp_node*( ))), *v*, *false*);
    mp_dep_mult(*mp*, (**mp_value_node**) *y_part*(*value_node*(*cur_exp_node*( ))), *v*, *false*);
  $\}$
  else if (*mp*→*cur_exp.type* ≡ *mp_color_type*) $\{$
    mp_dep_mult(*mp*, (**mp_value_node**) *red_part*(*value_node*(*cur_exp_node*( ))), *v*, *false*);
    mp_dep_mult(*mp*, (**mp_value_node**) *green_part*(*value_node*(*cur_exp_node*( ))), *v*, *false*);
    mp_dep_mult(*mp*, (**mp_value_node**) *blue_part*(*value_node*(*cur_exp_node*( ))), *v*, *false*);
  $\}$
  else if (*mp*→*cur_exp.type* ≡ *mp_cmykcolor_type*) $\{$
    mp_dep_mult(*mp*, (**mp_value_node**) *cyan_part*(*value_node*(*cur_exp_node*( ))), *v*, *false*);
    mp_dep_mult(*mp*, (**mp_value_node**) *magenta_part*(*value_node*(*cur_exp_node*( ))), *v*, *false*);
    mp_dep_mult(*mp*, (**mp_value_node**) *yellow_part*(*value_node*(*cur_exp_node*( ))), *v*, *false*);
    mp_dep_mult(*mp*, (**mp_value_node**) *black_part*(*value_node*(*cur_exp_node*( ))), *v*, *false*);
  $\}$
  else $\{$
    mp_dep_mult(*mp*, Λ, *v*, *false*);

```
    }
    if (old_exp ≠ Λ) {
        mp_recycle_value(mp, old_exp);
        mp_free_value_node(mp, old_exp);
    }
    free_number(v);
}
```

995.   ⟨Trace the fraction multiplication 995⟩ ≡

```
{
    mp_begin_diagnostic(mp);
    mp_print_nl(mp, "{(");
    print_number(n);
    mp_print_char(mp, xord('/'));
    print_number(d);
    mp_print(mp, ")*(");
    mp_print_exp(mp, Λ, 0);
    mp_print(mp, ")}");
    mp_end_diagnostic(mp, false);
}
```

This code is used in section 994.

996.   The *hard_times* routine multiplies a nice color or pair by a dependency list.

⟨ Declare binary action procedures 989 ⟩ +≡
  static void *mp_hard_times*(**MP** *mp*, **mp_node** *p*)
  {
    mp_value_node *q*;   /∗ a copy of the dependent variable *p* ∗/
    mp_value_node *pp*;   /∗ for typecasting p ∗/
    mp_node *r*;   /∗ a component of the big node for the nice color or pair ∗/
    mp_number *v*;   /∗ the known value for *r* ∗/
    new_number(*v*);
    if (*mp_type*(*p*) ≤ *mp_pair_type*) {
      q = (**mp_value_node**) *mp_stash_cur_exp*(*mp*);
      mp_unstash_cur_exp(*mp*, *p*);
      p = (**mp_node**) *q*;
    }   /∗ now *cur_type* = *mp_pair_type* or *cur_type* = *mp_color_type* or *cur_type* = *mp_cmykcolor_type* ∗/
    pp = (**mp_value_node**) *p*;
    if (*mp*→*cur_exp.type* ≡ *mp_pair_type*) {
      r = *x_part*(*value_node*(*cur_exp_node*()));
      number_clone(*v*, *value_number*(*r*));
      mp_new_dep(*mp*, *r*, *mp_type*(*pp*), *mp_copy_dep_list*(*mp*, (**mp_value_node**) *dep_list*(*pp*)));
      mp_dep_mult(*mp*, (**mp_value_node**) *r*, *v*, *true*);
      r = *y_part*(*value_node*(*cur_exp_node*()));
      number_clone(*v*, *value_number*(*r*));
      mp_new_dep(*mp*, *r*, *mp_type*(*pp*), *mp_copy_dep_list*(*mp*, (**mp_value_node**) *dep_list*(*pp*)));
      mp_dep_mult(*mp*, (**mp_value_node**) *r*, *v*, *true*);
    }
    else if (*mp*→*cur_exp.type* ≡ *mp_color_type*) {
      r = *red_part*(*value_node*(*cur_exp_node*()));
      number_clone(*v*, *value_number*(*r*));
      mp_new_dep(*mp*, *r*, *mp_type*(*pp*), *mp_copy_dep_list*(*mp*, (**mp_value_node**) *dep_list*(*pp*)));
      mp_dep_mult(*mp*, (**mp_value_node**) *r*, *v*, *true*);
      r = *green_part*(*value_node*(*cur_exp_node*()));
      number_clone(*v*, *value_number*(*r*));
      mp_new_dep(*mp*, *r*, *mp_type*(*pp*), *mp_copy_dep_list*(*mp*, (**mp_value_node**) *dep_list*(*pp*)));
      mp_dep_mult(*mp*, (**mp_value_node**) *r*, *v*, *true*);
      r = *blue_part*(*value_node*(*cur_exp_node*()));
      number_clone(*v*, *value_number*(*r*));
      mp_new_dep(*mp*, *r*, *mp_type*(*pp*), *mp_copy_dep_list*(*mp*, (**mp_value_node**) *dep_list*(*pp*)));
      mp_dep_mult(*mp*, (**mp_value_node**) *r*, *v*, *true*);
    }
    else if (*mp*→*cur_exp.type* ≡ *mp_cmykcolor_type*) {
      r = *cyan_part*(*value_node*(*cur_exp_node*()));
      number_clone(*v*, *value_number*(*r*));
      mp_new_dep(*mp*, *r*, *mp_type*(*pp*), *mp_copy_dep_list*(*mp*, (**mp_value_node**) *dep_list*(*pp*)));
      mp_dep_mult(*mp*, (**mp_value_node**) *r*, *v*, *true*);
      r = *yellow_part*(*value_node*(*cur_exp_node*()));
      number_clone(*v*, *value_number*(*r*));
      mp_new_dep(*mp*, *r*, *mp_type*(*pp*), *mp_copy_dep_list*(*mp*, (**mp_value_node**) *dep_list*(*pp*)));
      mp_dep_mult(*mp*, (**mp_value_node**) *r*, *v*, *true*);
      r = *magenta_part*(*value_node*(*cur_exp_node*()));
      number_clone(*v*, *value_number*(*r*));
      mp_new_dep(*mp*, *r*, *mp_type*(*pp*), *mp_copy_dep_list*(*mp*, (**mp_value_node**) *dep_list*(*pp*)));
      mp_dep_mult(*mp*, (**mp_value_node**) *r*, *v*, *true*);

$r = black\_part(value\_node(cur\_exp\_node( )));$
$number\_clone(v, value\_number(r));$
$mp\_new\_dep(mp, r, mp\_type(pp), mp\_copy\_dep\_list(mp, (\textbf{mp\_value\_node}) \; dep\_list(pp)));$
$mp\_dep\_mult(mp, (\textbf{mp\_value\_node}) \; r, v, true);$
    }
  $free\_number(v);$
  }

997.   ⟨Declare binary action procedures 989⟩ +≡
  static void $mp\_dep\_div(\textbf{MP} \; mp, \textbf{mp\_value\_node} \; p, \textbf{mp\_number} \; v)$
  {
    mp_value_node q;   /∗ the dependency list being divided by v ∗/
    quarterword s, t;   /∗ its type, before and after ∗/

    if $(p \equiv \Lambda)$ $q = (\textbf{mp\_value\_node}) \; cur\_exp\_node( );$
    else if $(mp\_type(p) \neq mp\_known)$ $q = p;$
    else {
      mp_number ret;

      $new\_number(ret);$
      $make\_scaled(ret, value\_number(p), v);$
      $set\_value\_number(p, ret);$
      $free\_number(ret);$
      return;
    }
    $t = mp\_type(q);$
    $q = (\textbf{mp\_value\_node}) \; dep\_list(q);$
    $s = t;$
    if $(t \equiv mp\_dependent)$ {
      mp_number ab_vs_cd;
      mp_number $arg1$, $arg2$;

      $new\_number(ab\_vs\_cd);$
      $new\_number(arg2);$
      $new\_fraction(arg1);$
      $mp\_max\_coef(mp, \&arg1, q);$
      $number\_clone(arg2, v);$
      $number\_abs(arg2);$
      $ab\_vs\_cd(ab\_vs\_cd, arg1, unity\_t, coef\_bound\_minus\_1, arg2);$
      $free\_number(arg1);$
      $free\_number(arg2);$
      if $(number\_nonnegative(ab\_vs\_cd))$ {
        $t = mp\_proto\_dependent;$
      }
      $free\_number(ab\_vs\_cd);$
    }
    $q = mp\_p\_over\_v(mp, q, v, s, t);$
    $mp\_dep\_finish(mp, q, p, t);$
  }

998.   Let c be one of the eight transform operators. The procedure call $set\_up\_trans(c)$ first changes
cur_exp to a transform that corresponds to c and the original value of cur_exp. (In particular, cur_exp
doesn't change at all if $c = transformed\_by$.)

Then, if all components of the resulting transform are $known$, they are moved to the global variables txx,
txy, tyx, tyy, tx, ty; and cur_exp is changed to the known value zero.

⟨ Declare binary action procedures 989 ⟩ +≡
```
  static void mp_set_up_trans(MP mp, quarterword c)
  {
    mp_node p, q, r;      /∗ list manipulation registers ∗/
    mp_value new_expr;

    memset(&new_expr, 0, sizeof(mp_value));
    if ((c ≠ mp_transformed_by) ∨ (mp→cur_exp.type ≠ mp_transform_type)) {
        /∗ Put the current transform into cur_exp ∗/
      const char ∗hlp[ ] = {"The␣expression␣shown␣above␣has␣the␣wrong␣type,",
          "so␣I␣can\'t␣transform␣anything␣using␣it.",
          "Proceed,␣and␣I'll␣omit␣the␣transformation.", Λ};
      p = mp_stash_cur_exp(mp);
      set_cur_exp_node(mp_id_transform(mp));
      mp→cur_exp.type = mp_transform_type;
      q = value_node(cur_exp_node( ));
      switch (c) {
        ⟨ For each of the eight cases, change the relevant fields of cur_exp and goto done; but do nothing
            if capsule p doesn't have the appropriate type 1002 ⟩;
      }
      ;     /∗ there are no other cases ∗/
      mp_disp_err(mp, p);
      mp_back_error(mp, "Improper␣transformation␣argument", hlp, true);
      mp_get_x_next(mp);
    DONE:  mp_recycle_value(mp, p);
      mp_free_value_node(mp, p);
    }     /∗ If the current transform is entirely known, stash it in global variables; otherwise return ∗/
    q = value_node(cur_exp_node( ));
    if (mp_type(tx_part(q)) ≠ mp_known) return;
    if (mp_type(ty_part(q)) ≠ mp_known) return;
    if (mp_type(xx_part(q)) ≠ mp_known) return;
    if (mp_type(xy_part(q)) ≠ mp_known) return;
    if (mp_type(yx_part(q)) ≠ mp_known) return;
    if (mp_type(yy_part(q)) ≠ mp_known) return;
    number_clone(mp→txx, value_number(xx_part(q)));
    number_clone(mp→txy, value_number(xy_part(q)));
    number_clone(mp→tyx, value_number(yx_part(q)));
    number_clone(mp→tyy, value_number(yy_part(q)));
    number_clone(mp→tx, value_number(tx_part(q)));
    number_clone(mp→ty, value_number(ty_part(q)));
    new_number(new_expr.data.n);
    set_number_to_zero(new_expr.data.n);
    mp_flush_cur_exp(mp, new_expr);
  }
```

999.   ⟨ Global variables 14 ⟩ +≡
  mp_number *txx*;
  mp_number *txy*;
  mp_number *tyx*;
  mp_number *tyy*;
  mp_number *tx*;
  mp_number *ty*;   /∗ current transform coefficients ∗/

1000.   ⟨ Initialize table entries 182 ⟩ +≡
  new_number(*mp*→*txx*);
  new_number(*mp*→*txy*);
  new_number(*mp*→*tyx*);
  new_number(*mp*→*tyy*);
  new_number(*mp*→*tx*);
  new_number(*mp*→*ty*);

1001.   ⟨ Free table entries 183 ⟩ +≡
  free_number(*mp*→*txx*);
  free_number(*mp*→*txy*);
  free_number(*mp*→*tyx*);
  free_number(*mp*→*tyy*);
  free_number(*mp*→*tx*);
  free_number(*mp*→*ty*);

1002. ⟨For each of the eight cases, change the relevant fields of *cur_exp* and **goto** *done*; but do nothing
if capsule *p* doesn't have the appropriate type 1002⟩ ≡

case *mp_rotated_by*:
  if (*mp_type*(*p*) ≡ *mp_known*) ⟨Install sines and cosines, then **goto** *done* 1003⟩;
  break;
case *mp_slanted_by*:
  if (*mp_type*(*p*) > *mp_pair_type*) {
    mp_install(*mp*, *xy_part*(*q*), *p*);
    goto DONE;
  }
  break;
case *mp_scaled_by*:
  if (*mp_type*(*p*) > *mp_pair_type*) {
    mp_install(*mp*, *xx_part*(*q*), *p*);
    mp_install(*mp*, *yy_part*(*q*), *p*);
    goto DONE;
  }
  break;
case *mp_shifted_by*:
  if (*mp_type*(*p*) ≡ *mp_pair_type*) {
    r = *value_node*(*p*);
    mp_install(*mp*, *tx_part*(*q*), *x_part*(*r*));
    mp_install(*mp*, *ty_part*(*q*), *y_part*(*r*));
    goto DONE;
  }
  break;
case *mp_x_scaled*:
  if (*mp_type*(*p*) > *mp_pair_type*) {
    mp_install(*mp*, *xx_part*(*q*), *p*);
    goto DONE;
  }
  break;
case *mp_y_scaled*:
  if (*mp_type*(*p*) > *mp_pair_type*) {
    mp_install(*mp*, *yy_part*(*q*), *p*);
    goto DONE;
  }
  break;
case *mp_z_scaled*:
  if (*mp_type*(*p*) ≡ *mp_pair_type*) ⟨Install a complex multiplier, then **goto** *done* 1004⟩;
  break;
case *mp_transformed_by*: **break**;

This code is used in section 998.

1003. ⟨Install sines and cosines, then **goto** *done* 1003⟩ ≡

```
{
  mp_number n_sin, n_cos, arg1, arg2;
  new_number(arg1);
  new_number(arg2);
  new_fraction(n_sin);
  new_fraction(n_cos);      /* results computed by n_sin_cos */
  number_clone(arg2, unity_t);
  number_clone(arg1, value_number(p));
  number_multiply_int(arg2, 360);
  number_modulo(arg1, arg2);
  convert_scaled_to_angle(arg1);
  n_sin_cos(arg1, n_cos, n_sin);
  fraction_to_round_scaled(n_sin);
  fraction_to_round_scaled(n_cos);
  set_value_number(xx_part(q), n_cos);
  set_value_number(yx_part(q), n_sin);
  set_value_number(xy_part(q), value_number(yx_part(q)));
  number_negate(value_number(xy_part(q)));
  set_value_number(yy_part(q), value_number(xx_part(q)));
  free_number(arg1);
  free_number(arg2);
  free_number(n_sin);
  free_number(n_cos);
  goto DONE;
}
```

This code is used in section 1002.

1004. ⟨Install a complex multiplier, then **goto** *done* 1004⟩ ≡

```
{
  r = value_node(p);
  mp_install(mp, xx_part(q), x_part(r));
  mp_install(mp, yy_part(q), x_part(r));
  mp_install(mp, yx_part(q), y_part(r));
  if (mp_type(y_part(r)) ≡ mp_known) {
    set_value_number(y_part(r), value_number(y_part(r)));
    number_negate(value_number(y_part(r)));
  }
  else {
    mp_negate_dep_list(mp, (mp_value_node) dep_list((mp_value_node) y_part(r)));
  }
  mp_install(mp, xy_part(q), y_part(r));
  goto DONE;
}
```

This code is used in section 1002.

1005.   Procedure *set_up_known_trans* is like *set_up_trans*, but it insists that the transformation be entirely known.

⟨ Declare binary action procedures 989 ⟩ +≡
  static void *mp_set_up_known_trans*(**MP** *mp*, **quarterword** *c*)
  {
    mp_set_up_trans(*mp*, *c*);
    if (*mp*→*cur_exp.type* ≠ *mp_known*) {
      mp_value *new_expr*;
      const char *∗hlp*[ ] = {"I'm␣unable␣to␣apply␣a␣partially␣specified␣transformation",
          "except␣to␣a␣fully␣known␣pair␣or␣transform.",
          "Proceed,␣and␣I'll␣omit␣the␣transformation.", Λ};
      memset(&*new_expr*, 0, **sizeof**(**mp_value**));
      new_number(*new_expr.data.n*);
      mp_disp_err(*mp*, Λ);
      set_number_to_zero(*new_expr.data.n*);
      mp_back_error(*mp*, "Transform␣components␣aren't␣all␣known", *hlp*, *true*);
      mp_get_x_next(*mp*);
      mp_flush_cur_exp(*mp*, *new_expr*);
      set_number_to_unity(*mp*→*txx*);
      set_number_to_zero(*mp*→*txy*);
      set_number_to_zero(*mp*→*tyx*);
      set_number_to_unity(*mp*→*tyy*);
      set_number_to_zero(*mp*→*tx*);
      set_number_to_zero(*mp*→*ty*);
    }
  }

1006.   Here's a procedure that applies the transform *txx* .. *ty* to a pair of coordinates in locations *p* and *q*.

⟨ Declare binary action procedures 989 ⟩ +≡
  static void *mp_number_trans*(**MP** *mp*, **mp_number** *∗p*, **mp_number** *∗q*)
  {
    mp_number *r1*, *r2*, *v*;
    new_number(*r1*);
    new_number(*r2*);
    new_number(*v*);
    take_scaled(*r1*, *∗p*, *mp*→*txx*);
    take_scaled(*r2*, *∗q*, *mp*→*txy*);
    number_add(*r1*, *r2*);
    set_number_from_addition(*v*, *r1*, *mp*→*tx*);
    take_scaled(*r1*, *∗p*, *mp*→*tyx*);
    take_scaled(*r2*, *∗q*, *mp*→*tyy*);
    number_add(*r1*, *r2*);
    set_number_from_addition(*∗q*, *r1*, *mp*→*ty*);
    number_clone(*∗p*, *v*);
    free_number(*r1*);
    free_number(*r2*);
    free_number(*v*);
  }

1007.   The simplest transformation procedure applies a transform to all coordinates of a path. The *path_trans*(*c*)(*p*) macro applies a transformation defined by *cur_exp* and the transform operator *c* to the path *p*.

#define *path_trans*(A, B)

```
    {
        mp_set_up_known_trans(mp, (A));
        mp_unstash_cur_exp(mp, (B));
        mp_do_path_trans(mp, cur_exp_knot( ));
    }
```

⟨ Declare binary action procedures 989 ⟩ +≡
 static void *mp_do_path_trans*(**MP** *mp*, **mp_knot** *p*)
 {
  mp_knot *q*; /∗ list traverser ∗/

  q = *p*;
  do {
   if (*mp_left_type*(*q*) ≠ *mp_endpoint*) *mp_number_trans*(*mp*, &*q*→*left_x*, &*q*→*left_y*);
   mp_number_trans(*mp*, &*q*→*x_coord*, &*q*→*y_coord*);
   if (*mp_right_type*(*q*) ≠ *mp_endpoint*) *mp_number_trans*(*mp*, &*q*→*right_x*, &*q*→*right_y*);
   q = *mp_next_knot*(*q*);
  } **while** (*q* ≠ *p*);
 }

1008.   Transforming a pen is very similar, except that there are no *mp_left_type* and *mp_right_type* fields.

#define *pen_trans*(A, B)

```
    {
        mp_set_up_known_trans(mp, (A));
        mp_unstash_cur_exp(mp, (B));
        mp_do_pen_trans(mp, cur_exp_knot( ));
    }
```

⟨ Declare binary action procedures 989 ⟩ +≡
 static void *mp_do_pen_trans*(**MP** *mp*, **mp_knot** *p*)
 {
  mp_knot *q*; /∗ list traverser ∗/

  if (*pen_is_elliptical*(*p*)) {
   mp_number_trans(*mp*, &*p*→*left_x*, &*p*→*left_y*);
   mp_number_trans(*mp*, &*p*→*right_x*, &*p*→*right_y*);
  }
  q = *p*;
  do {
   mp_number_trans(*mp*, &*q*→*x_coord*, &*q*→*y_coord*);
   q = *mp_next_knot*(*q*);
  } **while** (*q* ≠ *p*);
 }

1009.   The next transformation procedure applies to edge structures. It will do any transformation, but the results may be substandard if the picture contains text that uses downloaded bitmap fonts. The binary action procedure is *do_edges_trans*, but we also need a function that just scales a picture. That routine is *scale_edges*. Both it and the underlying routine *edges_trans* should be thought of as procedures that update an edge structure h, except that they have to return a (possibly new) structure because of the need to call *private_edges*.

⟨ Declare binary action procedures 989 ⟩ +≡

  static mp_edge_header_node *mp_edges_trans*(**MP** *mp*, **mp_edge_header_node** *h*)

  {

    mp_node *q*;   /∗ the object being transformed ∗/

    mp_dash_node *r*, *s*;   /∗ for list manipulation ∗/

    mp_number *sx*, *sy*;   /∗ saved transformation parameters ∗/

    mp_number *sqdet*;   /∗ square root of determinant for *dash_scale* ∗/

    mp_number *sgndet*;   /∗ sign of the determinant ∗/

    h = *mp_private_edges*(*mp*, *h*);

    new_number(*sx*);

    new_number(*sy*);

    new_number(*sqdet*);

    new_number(*sgndet*);

    mp_sqrt_det(*mp*, &*sqdet*, *mp*→*txx*, *mp*→*txy*, *mp*→*tyx*, *mp*→*tyy*);

    ab_vs_cd(*sgndet*, *mp*→*txx*, *mp*→*tyy*, *mp*→*txy*, *mp*→*tyx*);

    if (*dash_list*(*h*) ≠ *mp*→*null_dash*) {

      ⟨ Try to transform the dash list of *h* 1010 ⟩;

    }

    ⟨ Make the bounding box of *h* unknown if it can't be updated properly without scanning the whole

        structure 1013 ⟩;

    q = *mp_link*(*edge_list*(*h*));

    while (*q* ≠ Λ) {

      ⟨ Transform graphical object *q* 1016 ⟩;

      q = *mp_link*(*q*);

    }

    free_number(*sx*);

    free_number(*sy*);

    free_number(*sqdet*);

    free_number(*sgndet*);

    return *h*;

  }

  static void *mp_do_edges_trans*(**MP** *mp*, **mp_node** *p*, **quarterword** *c*)

  {

    mp_set_up_known_trans(*mp*, *c*);

    set_value_node(*p*, (**mp_node**) *mp_edges_trans*(*mp*, (**mp_edge_header_node**) *value_node*(*p*)));

    mp_unstash_cur_exp(*mp*, *p*);

  }

  static mp_edge_header_node *mp_scale_edges*(**MP** *mp*, **mp_number** *se_sf*, **mp_edge_header_node**

      se_pic)

  {

    number_clone(*mp*→*txx*, *se_sf*);

    number_clone(*mp*→*tyy*, *se_sf*);

    set_number_to_zero(*mp*→*txy*);

    set_number_to_zero(*mp*→*tyx*);

    set_number_to_zero(*mp*→*tx*);

```
    set_number_to_zero(mp→ty);
    return mp_edges_trans(mp, se_pic);
  }
```

1010. ⟨Try to transform the dash list of h 1010⟩ ≡
```
  if (number_nonzero(mp→txy) ∨ number_nonzero(mp→tyx) ∨ number_nonzero(mp→ty) ∨
        number_nonequalabs(mp→txx, mp→tyy)) {
    mp_flush_dash_list(mp, h);
  }
  else {
    mp_number abs_tyy, ret;
    new_number(abs_tyy);
    if (number_negative(mp→txx)) {
      ⟨Reverse the dash list of $h$ 1011⟩;
    }
    ⟨Scale the dash list by txx and shift it by tx 1012⟩;
    number_clone(abs_tyy, mp→tyy);
    number_abs(abs_tyy);
    new_number(ret);
    take_scaled(ret, h→dash_y, abs_tyy);
    number_clone(h→dash_y, ret);
    free_number(ret);
    free_number(abs_tyy);
  }
```
This code is used in section 1009.

1011. ⟨Reverse the dash list of h 1011⟩ ≡
```
  {
    r = dash_list(h);
    set_dash_list(h, mp→null_dash);
    while (r ≠ mp→null_dash) {
      s = r;
      r = (mp_dash_node) mp_link(r);
      number_swap(s→start_x, s→stop_x);
      mp_link(s) = (mp_node) dash_list(h);
      set_dash_list(h, s);
    }
  }
```
This code is used in section 1010.

1012.   ⟨Scale the dash list by *txx* and shift it by *tx* 1012⟩ ≡
  r = *dash_list*(*h*);
  {
    mp_number *arg1*;

    new_number(*arg1*);
    while (*r* ≠ *mp*→*null_dash*) {
      take_scaled(*arg1*, *r*→*start_x*, *mp*→*txx*);
      set_number_from_addition(*r*→*start_x*, *arg1*, *mp*→*tx*);
      take_scaled(*arg1*, *r*→*stop_x*, *mp*→*txx*);
      set_number_from_addition(*r*→*stop_x*, *arg1*, *mp*→*tx*);
      r = (**mp_dash_node**) *mp_link*(*r*);
    }
    free_number(*arg1*);
  }
This code is used in section 1010.

1013.   ⟨Make the bounding box of *h* unknown if it can't be updated properly without scanning the whole
      structure 1013⟩ ≡
  if (*number_zero*(*mp*→*txx*) ∧ *number_zero*(*mp*→*tyy*)) {
    ⟨Swap the *x* and *y* parameters in the bounding box of *h* 1014⟩;
  }
  else if (*number_nonzero*(*mp*→*txy*) ∨ *number_nonzero*(*mp*→*tyx*)) {
    mp_init_bbox(*mp*, *h*);
    goto DONE1;
  }
  if (*number_lessequal*(*h*→*minx*, *h*→*maxx*)) {
    ⟨Scale the bounding box by *txx* + *txy* and *tyx* + *tyy*; then shift by (*tx*, *ty*) 1015⟩;
  }
  DONE1:
This code is used in section 1009.

1014.   ⟨Swap the *x* and *y* parameters in the bounding box of *h* 1014⟩ ≡
  {
    number_swap(*h*→*minx*, *h*→*miny*);
    number_swap(*h*→*maxx*, *h*→*maxy*);
  }
This code is used in section 1013.

1015.   The sum "$txx + txy$" is whichever of txx or txy is nonzero. The other sum is similar.

⟨ Scale the bounding box by $txx + txy$ and $tyx + tyy$; then shift by (tx, ty) 1015 ⟩ ≡

  {

    mp_number *tot*, *ret*;

    new_number(*tot*);

    new_number(*ret*);

    set_number_from_addition(*tot*, *mp→txx*, *mp→txy*);

    take_scaled(*ret*, *h→minx*, *tot*);

    set_number_from_addition(*h→minx*, *ret*, *mp→tx*);

    take_scaled(*ret*, *h→maxx*, *tot*);

    set_number_from_addition(*h→maxx*, *ret*, *mp→tx*);

    set_number_from_addition(*tot*, *mp→tyx*, *mp→tyy*);

    take_scaled(*ret*, *h→miny*, *tot*);

    set_number_from_addition(*h→miny*, *ret*, *mp→ty*);

    take_scaled(*ret*, *h→maxy*, *tot*);

    set_number_from_addition(*h→maxy*, *ret*, *mp→ty*);

    set_number_from_addition(*tot*, *mp→txx*, *mp→txy*);

    if (*number_negative*(*tot*)) {

      number_swap(*h→minx*, *h→maxx*);

    }

    set_number_from_addition(*tot*, *mp→tyx*, *mp→tyy*);

    if (*number_negative*(*tot*)) {

      number_swap(*h→miny*, *h→maxy*);

    }

    free_number(*ret*);

    free_number(*tot*);

  }

This code is used in section 1013.

1016.   Now we ready for the main task of transforming the graphical objects in edge structure h.

$\langle$ Transform graphical object q $_{1016}$ $\rangle \equiv$

  switch $(mp\_type(q))$ {

  case $mp\_fill\_node\_type$:

    {

      mp_fill_node $qq = ($**mp_fill_node**$)$ q;

      $mp\_do\_path\_trans(mp, mp\_path\_p(qq))$;

      $\langle$ Transform $mp\_pen\_p(qq)$, making sure polygonal pens stay counter-clockwise $_{1017}$ $\rangle$;

    }

    break;

  case $mp\_stroked\_node\_type$:

    {

      mp_stroked_node $qq = ($**mp_stroked_node**$)$ q;

      $mp\_do\_path\_trans(mp, mp\_path\_p(qq))$;

      $\langle$ Transform $mp\_pen\_p(qq)$, making sure polygonal pens stay counter-clockwise $_{1017}$ $\rangle$;

    }

    break;

  case $mp\_start\_clip\_node\_type$: $mp\_do\_path\_trans(mp, mp\_path\_p(($**mp_start_clip_node**$)$ $q))$;

    break;

  case $mp\_start\_bounds\_node\_type$: $mp\_do\_path\_trans(mp, mp\_path\_p(($**mp_start_bounds_node**$)$ $q))$;

    break;

  case $mp\_text\_node\_type$: $\langle$ Transform the compact transformation $_{1018}$ $\rangle$;

    break;

  case $mp\_stop\_clip\_node\_type$: **case** $mp\_stop\_bounds\_node\_type$: **break**;

  default: /∗ there are no other valid cases, but please the compiler ∗/

    break;

  }

This code is used in section 1009.

1017.   Note that the shift parameters (tx, ty) apply only to the path being stroked. The *dash_scale* has to be adjusted to scale the dash lengths in *mp_dash_p*(*q*) since the PostScript output procedures will try to compensate for the transformation we are applying to *mp_pen_p*(*q*). Since this compensation is based on the square root of the determinant, *sqdet* is the appropriate factor.

We pass the mptrap test only if *dash_scale* is not adjusted, nowadays (backend is changed?)

$\langle$ Transform *mp_pen_p*(*qq*), making sure polygonal pens stay counter-clockwise 1017 $\rangle \equiv$
```
if (mp_pen_p(qq) ≠ Λ) {
    number_clone(sx, mp→tx);
    number_clone(sy, mp→ty);
    set_number_to_zero(mp→tx);
    set_number_to_zero(mp→ty);
    mp_do_pen_trans(mp, mp_pen_p(qq));
    if (number_nonzero(sqdet) ∧ ((mp_type(q) ≡ mp_stroked_node_type) ∧ (mp_dash_p(q) ≠ Λ))) {
        mp_number ret;

        new_number(ret);
        take_scaled(ret, ((mp_stroked_node) q)→dash_scale, sqdet);
        number_clone(((mp_stroked_node) q)→dash_scale, ret);
        free_number(ret);
    }
    if (¬pen_is_elliptical(mp_pen_p(qq)))
        if (number_negative(sgndet))
            mp_pen_p(qq) = mp_make_pen(mp, mp_copy_path(mp, mp_pen_p(qq)), true);
            /* this unreverses the pen */
    number_clone(mp→tx, sx);
    number_clone(mp→ty, sy);
}
```
This code is used in section 1016.

1018. $\langle$ Transform the compact transformation 1018 $\rangle \equiv$
```
mp_number_trans(mp, &((mp_text_node) q)→tx, &((mp_text_node) q)→ty);
number_clone(sx, mp→tx);
number_clone(sy, mp→ty);
set_number_to_zero(mp→tx);
set_number_to_zero(mp→ty);
mp_number_trans(mp, &((mp_text_node) q)→txx, &((mp_text_node) q)→tyx);
mp_number_trans(mp, &((mp_text_node) q)→txy, &((mp_text_node) q)→tyy);
number_clone(mp→tx, sx); number_clone(mp→ty, sy)
```
This code is used in section 1016.

1019.   The hard cases of transformation occur when big nodes are involved, and when some of their components are unknown.

⟨Declare binary action procedures 989 ⟩ +≡
  ⟨Declare subroutines needed by *big_trans* 1021 ⟩;

  static void mp_big_trans(**MP** mp, **mp_node** p, **quarterword** c)
  {
    mp_node q, r, pp, qq;   /∗ list manipulation registers ∗/
    $q = value\_node(p)$;
    if ($mp\_type(q) \equiv mp\_pair\_node\_type$) {
      if ($mp\_type(x\_part(q)) \neq mp\_known \vee mp\_type(y\_part(q)) \neq mp\_known$) {
        ⟨Transform an unknown big node and **return** 1020 ⟩;
      }
    }
    else {   /∗ $mp\_transform\_type$ ∗/
      if ($mp\_type(tx\_part(q)) \neq mp\_known \vee mp\_type(ty\_part(q)) \neq mp\_known \vee mp\_type(xx\_part(q)) \neq$
            $mp\_known \vee mp\_type(xy\_part(q)) \neq mp\_known \vee mp\_type(yx\_part(q)) \neq$
            $mp\_known \vee mp\_type(yy\_part(q)) \neq mp\_known$) {
        ⟨Transform an unknown big node and **return** 1020 ⟩;
      }
    }
    ⟨Transform a known big node 1022 ⟩;
  }   /∗ node p will now be recycled by do_binary ∗/

1020.   ⟨Transform an unknown big node and **return** 1020 ⟩ ≡
  {
    $mp\_set\_up\_known\_trans(mp, c)$;
    $mp\_make\_exp\_copy(mp, p)$;
    $r = value\_node(cur\_exp\_node())$;
    if ($mp\text{→}cur\_exp.type \equiv mp\_transform\_type$) {
      $mp\_bilin1(mp, yy\_part(r), mp\text{→}tyy, xy\_part(q), mp\text{→}tyx, zero\_t)$;
      $mp\_bilin1(mp, yx\_part(r), mp\text{→}tyy, xx\_part(q), mp\text{→}tyx, zero\_t)$;
      $mp\_bilin1(mp, xy\_part(r), mp\text{→}txx, yy\_part(q), mp\text{→}txy, zero\_t)$;
      $mp\_bilin1(mp, xx\_part(r), mp\text{→}txx, yx\_part(q), mp\text{→}txy, zero\_t)$;
    }
    $mp\_bilin1(mp, y\_part(r), mp\text{→}tyy, x\_part(q), mp\text{→}tyx, mp\text{→}ty)$;
    $mp\_bilin1(mp, x\_part(r), mp\text{→}txx, y\_part(q), mp\text{→}txy, mp\text{→}tx)$;
    return;
  }

This code is used in section 1019.

1021.   Let p point to a value field inside a big node of *cur_exp*, and let q point to a another value field. The *bilin1* procedure replaces p by $p \cdot t + q \cdot u + \delta$.

⟨ Declare subroutines needed by *big_trans* 1021 ⟩ ≡
  static void *mp_bilin1* (**MP** *mp*, **mp_node** *p*, **mp_number** *t*, **mp_node** *q*, **mp_number** *u*, **mp_number**
      delta_orig)
  {
    mp_number *delta*;

    new_number (*delta*);
    number_clone (*delta*, *delta_orig*);
    if (¬*number_equal* (*t*, *unity_t*)) {
      mp_dep_mult (*mp*, (**mp_value_node**) *p*, *t*, *true*);
    }
    if (*number_nonzero* (*u*)) {
      if (*mp_type* (*q*) ≡ *mp_known*) {
        mp_number *tmp*;

        new_number (*tmp*);
        take_scaled (*tmp*, *value_number* (*q*), *u*);
        number_add (*delta*, *tmp*);
        free_number (*tmp*);
      }
      else {   /∗ Ensure that *type* (*p*) = *mp_proto_dependent* ∗/
        if (*mp_type* (*p*) ≠ *mp_proto_dependent*) {
          if (*mp_type* (*p*) ≡ *mp_known*) {
            mp_new_dep (*mp*, *p*, *mp_type* (*p*), *mp_const_dependency* (*mp*, *value_number* (*p*)));
          }
          else {
            set_dep_list ((**mp_value_node**) *p*, *mp_p_times_v* (*mp*, (**mp_value_node**)
              dep_list ((**mp_value_node**) *p*), *unity_t*, *mp_dependent*, *mp_proto_dependent*, *true*));
          }
          mp_type (*p*) = *mp_proto_dependent*;
        }
        set_dep_list ((**mp_value_node**) *p*, *mp_p_plus_fq* (*mp*, (**mp_value_node**) *dep_list* ((**mp_value_node**)
          p), *u*, (**mp_value_node**) *dep_list* ((**mp_value_node**) *q*), *mp_proto_dependent*, *mp_type* (*q*)));
      }
    }
    if (*mp_type* (*p*) ≡ *mp_known*) {
      set_value_number (*p*, *value_number* (*p*));
      number_add (*value_number* (*p*), *delta*);
    }
    else {
      mp_number *tmp*;
      mp_value_node *r*;   /∗ list traverser ∗/

      new_number (*tmp*);
      r = (**mp_value_node**) *dep_list* ((**mp_value_node**) *p*);
      while (*dep_info* (*r*) ≠ Λ) *r* = (**mp_value_node**) *mp_link* (*r*);
      number_clone (*tmp*, *value_number* (*r*));
      number_add (*delta*, *tmp*);
      if (*r* ≠ (**mp_value_node**) *dep_list* ((**mp_value_node**) *p*)) *set_value_number* (*r*, *delta*);
      else {
        mp_recycle_value (*mp*, *p*);
        mp_type (*p*) = *mp_known*;

```
            set_value_number(p, delta);
        }
        free_number(tmp);
    }
    if (mp→fix_needed) mp_fix_dependencies(mp);
    free_number(delta);
}
```

See also sections 1023, 1024, and 1026.

This code is used in section 1019.

1022.   ⟨ Transform a known big node 1022 ⟩ ≡

```
mp_set_up_trans(mp, c);
if (mp→cur_exp.type ≡ mp_known) {
    ⟨ Transform known by known 1025 ⟩;
}
else {
    pp = mp_stash_cur_exp(mp);
    qq = value_node(pp);
    mp_make_exp_copy(mp, p);
    r = value_node(cur_exp_node());
    if (mp→cur_exp.type ≡ mp_transform_type) {
        mp_bilin2(mp, yy_part(r), yy_part(qq), value_number(xy_part(q)), yx_part(qq), Λ);
        mp_bilin2(mp, yx_part(r), yy_part(qq), value_number(xx_part(q)), yx_part(qq), Λ);
        mp_bilin2(mp, xy_part(r), xx_part(qq), value_number(yy_part(q)), xy_part(qq), Λ);
        mp_bilin2(mp, xx_part(r), xx_part(qq), value_number(yx_part(q)), xy_part(qq), Λ);
    }
    mp_bilin2(mp, y_part(r), yy_part(qq), value_number(x_part(q)), yx_part(qq), y_part(qq));
    mp_bilin2(mp, x_part(r), xx_part(qq), value_number(y_part(q)), xy_part(qq), x_part(qq));
    mp_recycle_value(mp, pp);
    mp_free_value_node(mp, pp);
}
```

This code is used in section 1019.

1023. Let p be a *mp_proto_dependent* value whose dependency list ends at *dep_final*. The following procedure adds v times another numeric quantity to p.

⟨Declare subroutines needed by *big_trans* 1021⟩ +≡

```
static void mp_add_mult_dep(MP mp, mp_value_node p, mp_number v, mp_node r)
{
  if (mp_type(r) ≡ mp_known) {
    mp_number ret;
    new_number(ret);
    take_scaled(ret, value_number(r), v);
    set_dep_value(mp→dep_final, dep_value(mp→dep_final));
    number_add(dep_value(mp→dep_final), ret);
    free_number(ret);
  }
  else {
    set_dep_list(p, mp_p_plus_fq(mp, (mp_value_node) dep_list(p), v, (mp_value_node)
        dep_list((mp_value_node) r), mp_proto_dependent, mp_type(r)));
    if (mp→fix_needed) mp_fix_dependencies(mp);
  }
}
```

1024. The *bilin2* procedure is something like *bilin1*, but with known and unknown quantities reversed. Parameter p points to a value field within the big node for *cur_exp*; and $type(p) = mp\_known$. Parameters t and u point to value fields elsewhere; so does parameter q, unless it is Λ (which stands for zero). Location p will be replaced by $p \cdot t + v \cdot u + q$.

⟨Declare subroutines needed by *big_trans* 1021⟩ +≡

```
static void mp_bilin2(MP mp, mp_node p, mp_node t, mp_number v, mp_node u, mp_node q)
{
  mp_number vv;      /* temporary storage for value(p) */
  new_number(vv);
  number_clone(vv, value_number(p));
  mp_new_dep(mp, p, mp_proto_dependent, mp_const_dependency(mp, zero_t));
     /* this sets dep_final */
  if (number_nonzero(vv)) {
    mp_add_mult_dep(mp, (mp_value_node) p, vv, t);      /* dep_final doesn't change */
  }
  if (number_nonzero(v)) {
    mp_number arg1;
    new_number(arg1);
    number_clone(arg1, v);
    mp_add_mult_dep(mp, (mp_value_node) p, arg1, u);
    free_number(arg1);
  }
  if (q ≠ Λ) mp_add_mult_dep(mp, (mp_value_node) p, unity_t, q);
  if (dep_list((mp_value_node) p) ≡ (mp_node) mp→dep_final) {
    number_clone(vv, dep_value(mp→dep_final));
    mp_recycle_value(mp, p);
    mp_type(p) = mp_known;
    set_value_number(p, vv);
  }
  free_number(vv);
}
```

1025. ⟨Transform known by known 1025⟩ ≡

```
{
    mp_make_exp_copy(mp, p);
    r = value_node(cur_exp_node( ));
    if (mp→cur_exp.type ≡ mp_transform_type) {
        mp_bilin3(mp, yy_part(r), mp→tyy, value_number(xy_part(q)), mp→tyx, zero_t);
        mp_bilin3(mp, yx_part(r), mp→tyy, value_number(xx_part(q)), mp→tyx, zero_t);
        mp_bilin3(mp, xy_part(r), mp→txx, value_number(yy_part(q)), mp→txy, zero_t);
        mp_bilin3(mp, xx_part(r), mp→txx, value_number(yx_part(q)), mp→txy, zero_t);
    }
    mp_bilin3(mp, y_part(r), mp→tyy, value_number(x_part(q)), mp→tyx, mp→ty);
    mp_bilin3(mp, x_part(r), mp→txx, value_number(y_part(q)), mp→txy, mp→tx);
}
```

This code is used in section 1022.

1026. Finally, in *bilin3* everything is *known*.

⟨Declare subroutines needed by *big_trans* 1021⟩ +≡

```
    static void mp_bilin3(MP mp, mp_node p, mp_number t, mp_number v, mp_number
            u, mp_number delta_orig)
{
    mp_number delta;
    mp_number tmp;

    new_number(tmp);
    new_number(delta);
    number_clone(delta, delta_orig);
    if (¬number_equal(t, unity_t)) {
        take_scaled(tmp, value_number(p), t);
    }
    else {
        number_clone(tmp, value_number(p));
    }
    number_add(delta, tmp);
    if (number_nonzero(u)) {
        mp_number ret;

        new_number(ret);
        take_scaled(ret, v, u);
        set_value_number(p, delta);
        number_add(value_number(p), ret);
        free_number(ret);
    }
    else set_value_number(p, delta);
    free_number(tmp);
    free_number(delta);
}
```

1027.   ⟨Declare binary action procedures 989⟩ +≡
  static void *mp_chop_path*(**MP** *mp*, **mp_node** *p*)
  {
    mp_knot *q*;   /* a knot in the original path */
    mp_knot *pp*, *qq*, *rr*, *ss*;   /* link variables for copies of path nodes */
    mp_number *a*, *b*;   /* indices for chopping */
    mp_number *l*;
    boolean *reversed*;   /* was $a > b$? */
    new_number(*a*);
    new_number(*b*);
    new_number(*l*);
    mp_path_length(*mp*, &*l*);
    number_clone(*a*, *value_number*(*x_part*(*p*)));
    number_clone(*b*, *value_number*(*y_part*(*p*)));
    if (*number_lessequal*(*a*, *b*)) {
      reversed = *false*;
    }
    else {
      reversed = *true*;
      number_swap(*a*, *b*);
    }   /* Dispense with the cases $a < 0$ and/or $b > l$ */
    if (*number_negative*(*a*)) {
      if (*mp_left_type*(*cur_exp_knot*()) ≡ *mp_endpoint*) {
        set_number_to_zero(*a*);
        if (*number_negative*(*b*)) *set_number_to_zero*(*b*);
      }
      else {
        do {
          number_add(*a*, *l*);
          number_add(*b*, *l*);
        } **while** (*number_negative*(*a*));   /* a cycle always has length $l > 0$ */
      }
    }
    if (*number_greater*(*b*, *l*)) {
      if (*mp_left_type*(*cur_exp_knot*()) ≡ *mp_endpoint*) {
        number_clone(*b*, *l*);
        if (*number_greater*(*a*, *l*)) *number_clone*(*a*, *l*);
      }
      else {
        while (*number_greaterequal*(*a*, *l*)) {
          number_substract(*a*, *l*);
          number_substract(*b*, *l*);
        }
      }
    }
    q = *cur_exp_knot*();
    while (*number_greaterequal*(*a*, *unity_t*)) {
      q = *mp_next_knot*(*q*);
      number_substract(*a*, *unity_t*);
      number_substract(*b*, *unity_t*);
    }
    if (*number_equal*(*b*, *a*)) {   /* Construct a path from *pp* to *qq* of length zero */

```
if (number_positive(a)) {
  mp_number arg1;

  new_number(arg1);
  number_clone(arg1, a);
  convert_scaled_to_fraction(arg1);
  mp_split_cubic(mp, q, arg1);
  free_number(arg1);
  q = mp_next_knot(q);
}
pp = mp_copy_knot(mp, q);
qq = pp;
}
else {      /* Construct a path from pp to qq of length ⌈b⌉ */
pp = mp_copy_knot(mp, q);
qq = pp;
do {
  q = mp_next_knot(q);
  rr = qq;
  qq = mp_copy_knot(mp, q);
  mp_next_knot(rr) = qq;
  number_substract(b, unity_t);
} while (number_positive(b));
if (number_positive(a)) {
  mp_number arg1;

  new_number(arg1);
  ss = pp;
  number_clone(arg1, a);
  convert_scaled_to_fraction(arg1);
  mp_split_cubic(mp, ss, arg1);
  free_number(arg1);
  pp = mp_next_knot(ss);
  mp_toss_knot(mp, ss);
  if (rr ≡ ss) {
    mp_number arg1, arg2;

    new_number(arg1);
    new_number(arg2);
    set_number_from_substraction(arg1, unity_t, a);
    number_clone(arg2, b);
    make_scaled(b, arg2, arg1);
    free_number(arg1);
    free_number(arg2);
    rr = pp;
  }
}
if (number_negative(b)) {
  mp_number arg1;

  new_number(arg1);
  set_number_from_addition(arg1, b, unity_t);
  convert_scaled_to_fraction(arg1);
  mp_split_cubic(mp, rr, arg1);
  free_number(arg1);
```

```
        mp_toss_knot(mp, qq);
        qq = mp_next_knot(rr);
      }
    }
    mp_left_type(pp) = mp_endpoint;
    mp_right_type(qq) = mp_endpoint;
    mp_next_knot(qq) = pp;
    mp_toss_knot_list(mp, cur_exp_knot());
    if (reversed) {
      set_cur_exp_knot(mp_next_knot(mp_htap_ypoc(mp, pp)));
      mp_toss_knot_list(mp, pp);
    }
    else {
      set_cur_exp_knot(pp);
    }
    free_number(l);
    free_number(a);
    free_number(b);
  }
```

1028.   ⟨Declare binary action procedures 989⟩ +≡

  static void *mp_set_up_offset*(**MP** *mp*, **mp_node** *p*)

  {

    mp_find_offset(*mp*, *value_number*(*x_part*(*p*)), *value_number*(*y_part*(*p*)), *cur_exp_knot*( ));

    mp_pair_value(*mp*, *mp*→*cur_x*, *mp*→*cur_y*);

  }

  static void *mp_set_up_direction_time*(**MP** *mp*, **mp_node** *p*)

  {

    mp_value *new_expr*;

    memset(&*new_expr*, 0, **sizeof**(**mp_value**));

    new_number(*new_expr.data.n*);

    mp_find_direction_time(*mp*, &*new_expr.data.n*, *value_number*(*x_part*(*p*)), *value_number*(*y_part*(*p*)),

      cur_exp_knot( ));

    mp_flush_cur_exp(*mp*, *new_expr*);

  }

  static void *mp_set_up_envelope*(**MP** *mp*, **mp_node** *p*)

  {

    unsigned char *ljoin*, *lcap*;

    mp_number *miterlim*;

    mp_knot *q* = *mp_copy_path*(*mp*, *cur_exp_knot*( ));   /∗ the original path ∗/

    new_number(*miterlim*);   /∗ TODO: accept elliptical pens for straight paths ∗/

    if (*pen_is_elliptical*(*value_knot*(*p*))) {

      mp_bad_envelope_pen(*mp*);

      set_cur_exp_knot(*q*);

      mp→*cur_exp.type* = *mp_path_type*;

      return;

    }

    if (*number_greater*(*internal_value*(*mp_linejoin*), *unity_t*)) *ljoin* = 2;

    else if (*number_positive*(*internal_value*(*mp_linejoin*))) *ljoin* = 1;

    else *ljoin* = 0;

    if (*number_greater*(*internal_value*(*mp_linecap*), *unity_t*)) *lcap* = 2;

    else if (*number_positive*(*internal_value*(*mp_linecap*))) *lcap* = 1;

    else *lcap* = 0;

    if (*number_less*(*internal_value*(*mp_miterlimit*), *unity_t*)) *set_number_to_unity*(*miterlim*);

    else *number_clone*(*miterlim*, *internal_value*(*mp_miterlimit*));

    set_cur_exp_knot(*mp_make_envelope*(*mp*, *q*, *value_knot*(*p*), *ljoin*, *lcap*, *miterlim*));

    mp→*cur_exp.type* = *mp_path_type*;

  }

1029.   This is pretty straightfoward. The one silly thing is that the output of $mp\_ps\_do\_font\_charstring$ has to be un-exported.

⟨Declare binary action procedures 989⟩ +≡

```
static void mp_set_up_glyph_infont(MP mp, mp_node p)
{
  mp_edge_object *h = Λ;
  mp_ps_font *f = Λ;

  char *n = mp_str(mp, cur_exp_str());

  f = mp_ps_font_parse(mp, (int) mp_find_font(mp, n));
  if (f ≠ Λ) {
    if (mp_type(p) ≡ mp_known) {
      int v = round_unscaled(value_number(p));

      if (v < 0 ∨ v > 255) {
        char msg[256];

        mp_snprintf(msg, 256, "glyph␣index␣too␣high␣(%d)", v);
        mp_error(mp, msg, Λ, true);
      }
      else {
        h = mp_ps_font_charstring(mp, f, v);
      }
    }
    else {
      n = mp_str(mp, value_str(p));
      h = mp_ps_do_font_charstring(mp, f, n);
    }
    mp_ps_font_free(mp, f);
  }
  if (h ≠ Λ) {
    set_cur_exp_node((mp_node) mp_gr_import(mp, h));
  }
  else {
    set_cur_exp_node((mp_node) mp_get_edge_header_node(mp));
    mp_init_edges(mp, (mp_edge_header_node) cur_exp_node());
  }
  mp→cur_exp.type = mp_picture_type;
}
```

1030. ⟨Declare binary action procedures 989⟩ +≡

static void *mp_find_point*(**MP** *mp*, **mp_number** *v_orig*, **quarterword** *c*)
{
    mp_knot *p*;　　/∗ the path ∗/
    mp_number *n*;　　/∗ its length ∗/
    mp_number *v*;
    new_number(*v*);
    new_number(*n*);
    number_clone(*v*, *v_orig*);
    p = *cur_exp_knot*( );
    if (*mp_left_type*(*p*) ≡ *mp_endpoint*) {
      set_number_to_unity(*n*);
      number_negate(*n*);
    }
    else {
      set_number_to_zero(*n*);
    }
    do {
      p = *mp_next_knot*(*p*);
      number_add(*n*, *unity_t*);
    } **while** (*p* ≠ *cur_exp_knot*( ));
    if (*number_zero*(*n*)) {
      set_number_to_zero(*v*);
    }
    else if (*number_negative*(*v*)) {
      if (*mp_left_type*(*p*) ≡ *mp_endpoint*) {
        set_number_to_zero(*v*);
      }
      else {　　/∗ $v = n - 1 - ((-v - 1) \% n) \equiv -((-v - 1) \% n) - 1 + n$ ∗/
        number_negate(*v*);
        number_add_scaled(*v*, −1);
        number_modulo(*v*, *n*);
        number_negate(*v*);
        number_add_scaled(*v*, −1);
        number_add(*v*, *n*);
      }
    }
    else if (*number_greater*(*v*, *n*)) {
      if (*mp_left_type*(*p*) ≡ *mp_endpoint*) *number_clone*(*v*, *n*);
      else *number_modulo*(*v*, *n*);
    }
    p = *cur_exp_knot*( );
    while (*number_greaterequal*(*v*, *unity_t*)) {
      p = *mp_next_knot*(*p*);
      number_substract(*v*, *unity_t*);
    }
    if (*number_nonzero*(*v*)) {　　/∗ Insert a fractional node by splitting the cubic ∗/
      convert_scaled_to_fraction(*v*);
      mp_split_cubic(*mp*, *p*, *v*);
      p = *mp_next_knot*(*p*);
    }　　/∗ Set the current expression to the desired path coordinates ∗/
    switch (*c*) {

```
    case mp_point_of : mp_pair_value(mp, p→x_coord, p→y_coord);
      break;
    case mp_precontrol_of :
      if (mp_left_type(p) ≡ mp_endpoint) mp_pair_value(mp, p→x_coord, p→y_coord);
      else  mp_pair_value(mp, p→left_x, p→left_y);
      break;
    case mp_postcontrol_of :
      if (mp_right_type(p) ≡ mp_endpoint) mp_pair_value(mp, p→x_coord, p→y_coord);
      else  mp_pair_value(mp, p→right_x, p→right_y);
      break;
    }      /* there are no other cases */
    free_number(v);
    free_number(n);
  }
```

1031.   Function *new_text_node* owns the reference count for its second argument (the text string) but not its first (the font name).

⟨Declare binary action procedures 989⟩ +≡

```
  static void mp_do_infont(MP mp, mp_node p)
  {
    mp_edge_header_node q;
    mp_value new_expr;

    memset(&new_expr, 0, sizeof(mp_value));
    new_number(new_expr.data.n);
    q = mp_get_edge_header_node(mp);
    mp_init_edges(mp, q);
    add_str_ref(cur_exp_str( ));
    mp_link(obj_tail(q)) = mp_new_text_node(mp, mp_str(mp, cur_exp_str( )), value_str(p));
    obj_tail(q) = mp_link(obj_tail(q));
    mp_free_value_node(mp, p);
    new_expr.data.node = (mp_node) q;
    mp_flush_cur_exp(mp, new_expr);
    mp→cur_exp.type = mp_picture_type;
  }
```

1032.   Statements and commands.   The chief executive of METAPOST is the *do_statement* routine, which contains the master switch that causes all the various pieces of METAPOST to do their things, in the right order.

In a sense, this is the grand climax of the program: It applies all the tools that we have worked so hard to construct. In another sense, this is the messiest part of the program: It necessarily refers to other pieces of code all over the place, so that a person can't fully understand what is going on without paging back and forth to be reminded of conventions that are defined elsewhere. We are now at the hub of the web.

The structure of *do_statement* itself is quite simple. The first token of the statement is fetched using *get_x_next*. If it can be the first token of an expression, we look for an equation, an assignment, or a title. Otherwise we use a **case** construction to branch at high speed to the appropriate routine for various and sundry other types of commands, each of which has an "action procedure" that does the necessary work.

The program uses the fact that

$$min\_primary\_command = max\_statement\_command = type\_name$$

to interpret a statement that starts with, e.g., '**string**', as a type declaration rather than a boolean expression.

```
static void worry_about_bad_statement(MP mp);
static void flush_unparsable_junk_after_statement(MP mp);

void mp_do_statement(MP mp)
{      /* governs METAPOST's activities */
  mp→cur_exp.type = mp_vacuous;
  mp_get_x_next(mp);
  if (cur_cmd( ) > mp_max_primary_command) {
    worry_about_bad_statement(mp);
  }
  else if (cur_cmd( ) > mp_max_statement_command) {      /* Do an equation, assignment, title, or
        '⟨expression⟩ endgroup'; */      /* The most important statements begin with expressions */
    mp_value new_expr;

    mp→var_flag = mp_assignment;
    mp_scan_expression(mp);
    if (cur_cmd( ) < mp_end_group) {
      if (cur_cmd( ) ≡ mp_equals) mp_do_equation(mp);
      else if (cur_cmd( ) ≡ mp_assignment) mp_do_assignment(mp);
      else if (mp→cur_exp.type ≡ mp_string_type) {      /* Do a title */
        if (number_positive(internal_value(mp_tracing_titles))) {
          mp_print_nl(mp, "");
          mp_print_str(mp, cur_exp_str( ));
          update_terminal( );
        }
      }
    }
    else if (mp→cur_exp.type ≠ mp_vacuous) {
      const char *hlp[ ] = {"I␣couldn't␣find␣an␣'='␣or␣':='␣after␣the",
          "expression␣that␣is␣shown␣above␣this␣error␣message,",
          "so␣I␣guess␣I'll␣just␣ignore␣it␣and␣carry␣on.", Λ};
      mp_disp_err(mp, Λ);
      mp_back_error(mp, "Isolated␣expression", hlp, true);
      mp_get_x_next(mp);
    }
    memset(&new_expr, 0, sizeof(mp_value));
    new_number(new_expr.data.n);
    set_number_to_zero(new_expr.data.n);
```

```
        mp_flush_cur_exp(mp, new_expr);
        mp→cur_exp.type = mp_vacuous;
    }
}
else {        /∗ Do a statement that doesn't begin with an expression ∗/
        /∗ If do_statement ends with cur_cmd = end_group, we should have cur_type = mp_vacuous
          unless the statement was simply an expression; in the latter case, cur_type and cur_exp should
          represent that expression. ∗/
    if (number_positive(internal_value(mp_tracing_commands))) show_cur_cmd_mod;
    switch (cur_cmd()) {
    case mp_type_name: mp_do_type_declaration(mp);
        break;
    case mp_macro_def:
        if (cur_mod() > var_def) mp_make_op_def(mp);
        else if (cur_mod() > end_def) mp_scan_def(mp);
        break;
    case mp_random_seed: mp_do_random_seed(mp);
        break;
    case mp_mode_command: mp_print_ln(mp);
        mp→interaction = cur_mod();
        initialize_print_selector();
        if (mp→log_opened) mp→selector = mp→selector + 2;
        mp_get_x_next(mp);
        break;
    case mp_protection_command: mp_do_protection(mp);
        break;
    case mp_delimiters: mp_def_delims(mp);
        break;
    case mp_save_command: do {
            mp_get_symbol(mp);
            mp_save_variable(mp, cur_sym());
            mp_get_x_next(mp);
        } while (cur_cmd() ≡ mp_comma);
        break;
    case mp_interim_command: mp_do_interim(mp);
        break;
    case mp_let_command: mp_do_let(mp);
        break;
    case mp_new_internal: mp_do_new_internal(mp);
        break;
    case mp_show_command: mp_do_show_whatever(mp);
        break;
    case mp_add_to_command: mp_do_add_to(mp);
        break;
    case mp_bounds_command: mp_do_bounds(mp);
        break;
    case mp_ship_out_command: mp_do_ship_out(mp);
        break;
    case mp_every_job_command: mp_get_symbol(mp);
        mp→start_sym = cur_sym();
        mp_get_x_next(mp);
        break;
```

```
    case mp_message_command: mp_do_message(mp);
      break;
    case mp_write_command: mp_do_write(mp);
      break;
    case mp_tfm_command: mp_do_tfm_command(mp);
      break;
    case mp_special_command:
      if (cur_mod() ≡ 0) mp_do_special(mp);
      else if (cur_mod() ≡ 1) mp_do_mapfile(mp);
      else mp_do_mapline(mp);
      break;
    default: break;      /* make the compiler happy */
    }
    mp→cur_exp.type = mp_vacuous;
  }
  if (cur_cmd() < mp_semicolon) flush_unparsable_junk_after_statement(mp);
  mp→error_count = 0;
}
```

1033. ⟨ Declarations 8 ⟩ +≡
  ⟨ Declare action procedures for use by *do_statement* 1048 ⟩

1034.   The only command codes > *max_primary_command* that can be present at the beginning of a
statement are *semicolon* and higher; these occur when the statement is null.

```
static void worry_about_bad_statement(MP mp)
{
  if (cur_cmd() < mp_semicolon) {
    char msg[256];
    mp_string sname;
    int old_setting = mp→selector;
    const char *hlp[] = {"I␣was␣looking␣for␣the␣beginning␣of␣a␣new␣statement.",
        "If␣you␣just␣proceed␣without␣changing␣anything,␣I'll␣ignore",
        "everything␣up␣to␣the␣next␣';'.␣Please␣insert␣a␣semicolon",
        "now␣in␣front␣of␣anything␣that␣you␣don't␣want␣me␣to␣delete.",
        "(See␣Chapter␣27␣of␣The␣METAFONTbook␣for␣an␣example.)", Λ};
    mp→selector = new_string;
    mp_print_cmd_mod(mp, cur_cmd(), cur_mod());
    sname = mp_make_string(mp);
    mp→selector = old_setting;
    mp_snprintf(msg, 256, "A␣statement␣can't␣begin␣with␣'%s'", mp_str(mp, sname));
    delete_str_ref(sname);
    mp_back_error(mp, msg, hlp, true);
    mp_get_x_next(mp);
  }
}
```

1035.   The help message printed here says that everything is flushed up to a semicolon, but actually the
commands *end_group* and *stop* will also terminate a statement.

> **static void** *flush_unparsable_junk_after_statement*(**MP** *mp*)
> {
>   **const char** ∗*hlp*[ ] = {"I've␣just␣read␣as␣much␣of␣that␣statement␣as␣I␣could␣fathom,",
>   "so␣a␣semicolon␣should␣have␣been␣next.␣It's␣very␣puzzling...",
>   "but␣I'll␣try␣to␣get␣myself␣back␣together,␣by␣ignoring",
>   "everything␣up␣to␣the␣next␣`;'.␣Please␣insert␣a␣semicolon",
>   "now␣in␣front␣of␣anything␣that␣you␣don't␣want␣me␣to␣delete.",
>   "(See␣Chapter␣27␣of␣The␣METAFONTbook␣for␣an␣example.)", Λ};
>
>   *mp_back_error*(*mp*, "Extra␣tokens␣will␣be␣flushed", *hlp*, *true*);
>   *mp*→*scanner_status* = *flushing*;
>   **do** {
>   *get_t_next*(*mp*);
>   **if** (*cur_cmd*( ) ≡ *mp_string_token*) {
>   *delete_str_ref*(*cur_mod_str*( ));
>   }
>   } **while** (¬*mp_end_of_statement*);   /∗ *cur_cmd* = *semicolon*, *end_group*, or *stop* ∗/
>   *mp*→*scanner_status* = *normal*;
> }

1036.   Equations and assignments are performed by the pair of mutually recursive routines *do_equation*
and *do_assignment*. These routines are called when *cur_cmd* = *equals* and when *cur_cmd* = *assignment*,
respectively; the left-hand side is in *cur_type* and *cur_exp*, while the right-hand side is yet to be scanned.
After the routines are finished, *cur_type* and *cur_exp* will be equal to the right-hand side (which will normally
be equal to the left-hand side).

⟨ Declarations 8 ⟩ +≡
  ⟨ Declare the procedure called *make_eq* 1040 ⟩;
  static void *mp_do_equation*(**MP** *mp*);

1037.   **static void** *trace_equation*(**MP** *mp*, **mp_node** *lhs*)
{
  mp_begin_diagnostic(*mp*);
  mp_print_nl(*mp*, "{(");
  mp_print_exp(*mp*, *lhs*, 0);
  mp_print(*mp*, ")=(");
  mp_print_exp(*mp*, Λ, 0);
  mp_print(*mp*, ")}");
  mp_end_diagnostic(*mp*, *false*);
}
void *mp_do_equation*(**MP** *mp*)
{
  mp_node *lhs*;   /∗ capsule for the left-hand side ∗/

  lhs = *mp_stash_cur_exp*(*mp*);
  mp_get_x_next(*mp*);
  mp→*var_flag* = *mp_assignment*;
  mp_scan_expression(*mp*);
  if (*cur_cmd*( ) ≡ *mp_equals*) *mp_do_equation*(*mp*);
  else if (*cur_cmd*( ) ≡ *mp_assignment*) *mp_do_assignment*(*mp*);
  if (*number_greater*(*internal_value*(*mp_tracing_commands*), *two_t*)) {
    trace_equation(*mp*, *lhs*);
  }
  if (*mp*→*cur_exp.type* ≡ *mp_unknown_path*) {
    if (*mp_type*(*lhs*) ≡ *mp_pair_type*) {
      mp_node *p*;   /∗ temporary register ∗/

      p = *mp_stash_cur_exp*(*mp*);
      mp_unstash_cur_exp(*mp*, *lhs*);
      lhs = *p*;
    }   /∗ in this case *make_eq* will change the pair to a path ∗/
  }
  mp_make_eq(*mp*, *lhs*);   /∗ equate *lhs* to (*cur_type*, *cur_exp*) ∗/
}

1038.   And *do_assignment* is similar to *do_equation*:

⟨ Declarations 8 ⟩ +≡
  static void *mp_do_assignment*(**MP** *mp*);

1039.   static void *bad_lhs*(**MP** *mp*)
{
  const char ∗*hlp*[ ] = {"I␣didn't␣find␣a␣variable␣name␣at␣the␣left␣of␣the␣`:=',",
      "so␣I'm␣going␣to␣pretend␣that␣you␣said␣`='␣instead.", Λ};
  mp_disp_err(*mp*, Λ);
  mp_error(*mp*, "Improper␣`:='␣will␣be␣changed␣to␣`='", *hlp*, *true*);
  mp_do_equation(*mp*);
}
static void *bad_internal_assignment*(**MP** *mp*, **mp_node** *lhs*)
{
  char *msg*[256];
  const char ∗*hlp*[ ] = {"I␣can\'t␣set␣this␣internal␣quantity␣to␣anything␣but␣a␣known",
      "numeric␣value,␣so␣I'll␣have␣to␣ignore␣this␣assignment.", Λ};
  mp_disp_err(*mp*, Λ);
  if (*internal_type*(*mp_sym_info*(*lhs*)) ≡ *mp_known*) {
    mp_snprintf(*msg*, 256, "Internal␣quantity␣`%s'␣must␣receive␣a␣known␣numeric␣value",
        internal_name(*mp_sym_info*(*lhs*)));
  }
  else {
    mp_snprintf(*msg*, 256, "Internal␣quantity␣`%s'␣must␣receive␣a␣known␣string",
        internal_name(*mp_sym_info*(*lhs*)));
    hlp[1] = "string,␣so␣I'll␣have␣to␣ignore␣this␣assignment.";
  }
  mp_back_error(*mp*, *msg*, *hlp*, *true*);
  mp_get_x_next(*mp*);
}
static void *forbidden_internal_assignment*(**MP** *mp*, **mp_node** *lhs*)
{
  char *msg*[256];
  const char ∗*hlp*[ ] = {"I␣can\'t␣set␣this␣internal␣quantity␣to␣anything␣just␣yet",
      "(it␣is␣read-only),␣so␣I'll␣have␣to␣ignore␣this␣assignment.", Λ};
  mp_snprintf(*msg*, 256, "Internal␣quantity␣`%s'␣is␣read-only", *internal_name*(*mp_sym_info*(*lhs*)));
  mp_back_error(*mp*, *msg*, *hlp*, *true*);
  mp_get_x_next(*mp*);
}
static void *bad_internal_assignment_precision*(**MP** *mp*, **mp_node** *lhs*, **mp_number** *min*, **mp_number**
        max)
{
  char *msg*[256];
  char *s*[256];
  const char ∗*hlp*[ ] = {"Precision␣values␣are␣limited␣by␣the␣current␣numbersystem.", Λ, Λ};
  mp_snprintf(*msg*, 256, "Bad␣`%s'␣has␣been␣ignored", *internal_name*(*mp_sym_info*(*lhs*)));
  mp_snprintf(*s*, 256, "Currently␣I␣am␣using␣`%s';␣the␣allowed␣precision␣range␣is␣[%s,%s].",
      mp_str(*mp*, *internal_string*(*mp_number_system*)), *number_tostring*(*min*), *number_tostring*(*max*));
  hlp[1] = *s*;
  mp_back_error(*mp*, *msg*, *hlp*, *true*);
  mp_get_x_next(*mp*);
}
static void *bad_expression_assignment*(**MP** *mp*, **mp_node** *lhs*)
{

```
        const char *hlp[] = {"It␣seems␣you␣did␣a␣nasty␣thing---probably␣by␣accident,",
            "but␣nevertheless␣you␣nearly␣hornswoggled␣me...",
            "While␣I␣was␣evaluating␣the␣right-hand␣side␣of␣this",
            "command,␣something␣happened,␣and␣the␣left-hand␣side",
            "is␣no␣longer␣a␣variable!␣So␣I␣won't␣change␣anything.", Λ};
        char *msg = mp_obliterated(mp, lhs);
        mp_back_error(mp, msg, hlp, true);
        free(msg);
        mp_get_x_next(mp);
    }
    static void trace_assignment(MP mp, mp_node lhs)
    {
        mp_begin_diagnostic(mp);
        mp_print_nl(mp, "{");
        if (mp_name_type(lhs) ≡ mp_internal_sym) mp_print(mp, internal_name(mp_sym_info(lhs)));
        else mp_show_token_list(mp, lhs, Λ, 1000, 0);
        mp_print(mp, ":=");
        mp_print_exp(mp, Λ, 0);
        mp_print_char(mp, xord('}'));
        mp_end_diagnostic(mp, false);
    }
    void mp_do_assignment(MP mp)
    {
        if (mp→cur_exp.type ≠ mp_token_list) {
            bad_lhs(mp);
        }
        else {
            mp_node lhs;      /* token list for the left-hand side */
            lhs = cur_exp_node();
            mp→cur_exp.type = mp_vacuous;
            mp_get_x_next(mp);
            mp→var_flag = mp_assignment;
            mp_scan_expression(mp);
            if (cur_cmd() ≡ mp_equals) mp_do_equation(mp);
            else if (cur_cmd() ≡ mp_assignment) mp_do_assignment(mp);
            if (number_greater(internal_value(mp_tracing_commands), two_t)) {
                trace_assignment(mp, lhs);
            }
            if (mp_name_type(lhs) ≡ mp_internal_sym) {
                /* Assign the current expression to an internal variable */
                if ((mp→cur_exp.type ≡ mp_known ∨ mp→cur_exp.type ≡
                        mp_string_type) ∧ (internal_type(mp_sym_info(lhs)) ≡ mp→cur_exp.type))
                    {
                    if (mp_sym_info(lhs) ≡ mp_number_system) {
                        forbidden_internal_assignment(mp, lhs);
                    }
                    else if (mp_sym_info(lhs) ≡ mp_number_precision) {
                        if (¬(mp→cur_exp.type ≡ mp_known ∧ (¬number_less(cur_exp_value_number(),
                                precision_min)) ∧ (¬number_greater(cur_exp_value_number(), precision_max)))) {
                            bad_internal_assignment_precision(mp, lhs, precision_min, precision_max);
                        }
```

```
          else {
             set_internal_from_cur_exp(mp_sym_info(lhs));
             set_precision();
          }
        }
        else {
          set_internal_from_cur_exp(mp_sym_info(lhs));
        }
      }
      else {
        bad_internal_assignment(mp, lhs);
      }
    }
    else {      /* Assign the current expression to the variable lhs */
      mp_node p;       /* where the left-hand value is stored */
      mp_node q;        /* temporary capsule for the right-hand value */

      p = mp_find_variable(mp, lhs);
      if (p ≠ Λ) {
        q = mp_stash_cur_exp(mp);
        mp⃗cur_exp.type = mp_und_type(mp, p);
        mp_recycle_value(mp, p);
        mp_type(p) = mp⃗cur_exp.type;
        set_value_number(p, zero_t);
        mp_make_exp_copy(mp, p);
        p = mp_stash_cur_exp(mp);
        mp_unstash_cur_exp(mp, q);
        mp_make_eq(mp, p);
      }
      else {
        bad_expression_assignment(mp, lhs);
      }
    }
    mp_flush_node_list(mp, lhs);
  }
}
```

1040.   And now we get to the nitty-gritty. The *make_eq* procedure is given a pointer to a capsule that is to be equated to the current expression.

⟨ Declare the procedure called *make_eq* 1040 ⟩ ≡
  static void *mp_make_eq*(**MP** *mp*, **mp_node** *lhs*);

This code is used in section 1036.

1041.

```
static void announce_bad_equation(MP mp, mp_node lhs)
{
  char msg[256];
  const char *hlp[] = {"I'm␣sorry,␣but␣I␣don't␣know␣how␣to␣make␣such␣things␣equal.",
      "(See␣the␣two␣expressions␣just␣above␣the␣error␣message.)", Λ};
  mp_snprintf(msg, 256, "Equation␣cannot␣be␣performed␣(%s=%s)",
      (mp_type(lhs) ≤ mp_pair_type ? mp_type_string(mp_type(lhs)) : "numeric"),
      (mp→cur_exp.type ≤ mp_pair_type ? mp_type_string(mp→cur_exp.type) : "numeric"));
  mp_disp_err(mp, lhs);
  mp_disp_err(mp, Λ);
  mp_back_error(mp, msg, hlp, true);
  mp_get_x_next(mp);
}

static void exclaim_inconsistent_equation(MP mp)
{
  const char *hlp[] = {"The␣equation␣I␣just␣read␣contradicts␣what␣was␣said␣before.",
      "But␣don't␣worry;␣continue␣and␣I'll␣just␣ignore␣it.", Λ};
  mp_back_error(mp, "Inconsistent␣equation", hlp, true);
  mp_get_x_next(mp);
}

static void exclaim_redundant_or_inconsistent_equation(MP mp)
{
  const char *hlp[] = {"An␣equation␣between␣already-known␣quantities␣can't␣help.",
      "But␣don't␣worry;␣continue␣and␣I'll␣just␣ignore␣it.", Λ};
  mp_back_error(mp, "Redundant␣or␣inconsistent␣equation", hlp, true);
  mp_get_x_next(mp);
}

static void report_redundant_or_inconsistent_equation(MP mp, mp_node lhs, mp_number v)
{
  if (mp→cur_exp.type ≤ mp_string_type) {
    if (mp→cur_exp.type ≡ mp_string_type) {
      if (mp_str_vs_str(mp, value_str(lhs), cur_exp_str()) ≠ 0) {
        exclaim_inconsistent_equation(mp);
      }
      else {
        exclaim_redundant_equation(mp);
      }
    }
    else if (¬number_equal(v, cur_exp_value_number())) {
      exclaim_inconsistent_equation(mp);
    }
    else {
      exclaim_redundant_equation(mp);
    }
  }
  else {
    exclaim_redundant_or_inconsistent_equation(mp);
  }
}
```

```
void mp_make_eq(MP mp, mp_node lhs)
{
   mp_value new_expr;
   mp_variable_type t;      /* type of the left-hand side */
   mp_number v;      /* value of the left-hand side */
   memset(&new_expr, 0, sizeof(mp_value));
   new_number(v);
RESTART: t = mp_type(lhs);
   if (t ≤ mp_pair_type) number_clone(v, value_number(lhs));
      /* For each type t, make an equation or complain if cur_type is incompatible with t */
   switch (t) {
   case mp_boolean_type: case mp_string_type: case mp_pen_type: case mp_path_type:
      case mp_picture_type:
      if (mp→cur_exp.type ≡ t + unknown_tag) {
         new_number(new_expr.data.n);
         if (t ≡ mp_boolean_type) {
            number_clone(new_expr.data.n, v);
         }
         else if (t ≡ mp_string_type) {
            new_expr.data.str = value_str(lhs);
         }
         else if (t ≡ mp_picture_type) {
            new_expr.data.node = value_node(lhs);
         }
         else {      /* pen or path */
            new_expr.data.p = value_knot(lhs);
         }
         mp_nonlinear_eq(mp, new_expr, cur_exp_node( ), false);
         mp_unstash_cur_exp(mp, cur_exp_node( ));
      }
      else if (mp→cur_exp.type ≡ t) {
         report_redundant_or_inconsistent_equation(mp, lhs, v);
      }
      else {
         announce_bad_equation(mp, lhs);
      }
      break;
   case unknown_types:
      if (mp→cur_exp.type ≡ t − unknown_tag) {
         mp_nonlinear_eq(mp, mp→cur_exp, lhs, true);
      }
      else if (mp→cur_exp.type ≡ t) {
         mp_ring_merge(mp, lhs, cur_exp_node( ));
      }
      else if (mp→cur_exp.type ≡ mp_pair_type) {
         if (t ≡ mp_unknown_path) {
            mp_pair_to_path(mp);
            goto RESTART;
         }
      }
      else {
```

```
      announce_bad_equation(mp, lhs);
    }
  break;
case mp_transform_type: case mp_color_type: case mp_cmykcolor_type: case mp_pair_type:
  if (mp→cur_exp.type ≡ t) {        /* Do multiple equations */
    mp_node q = value_node(cur_exp_node());
    mp_node p = value_node(lhs);

    switch (t) {
    case mp_transform_type: mp_try_eq(mp, yy_part(p), yy_part(q));
      mp_try_eq(mp, yx_part(p), yx_part(q));
      mp_try_eq(mp, xy_part(p), xy_part(q));
      mp_try_eq(mp, xx_part(p), xx_part(q));
      mp_try_eq(mp, ty_part(p), ty_part(q));
      mp_try_eq(mp, tx_part(p), tx_part(q));
      break;
    case mp_color_type: mp_try_eq(mp, blue_part(p), blue_part(q));
      mp_try_eq(mp, green_part(p), green_part(q));
      mp_try_eq(mp, red_part(p), red_part(q));
      break;
    case mp_cmykcolor_type: mp_try_eq(mp, black_part(p), black_part(q));
      mp_try_eq(mp, yellow_part(p), yellow_part(q));
      mp_try_eq(mp, magenta_part(p), magenta_part(q));
      mp_try_eq(mp, cyan_part(p), cyan_part(q));
      break;
    case mp_pair_type: mp_try_eq(mp, y_part(p), y_part(q));
      mp_try_eq(mp, x_part(p), x_part(q));
      break;
    default:       /* there are no other valid cases, but please the compiler */
      break;
    }
  }
  else {
    announce_bad_equation(mp, lhs);
  }
  break;
case mp_known: case mp_dependent: case mp_proto_dependent: case mp_independent:
  if (mp→cur_exp.type ≥ mp_known) {
    mp_try_eq(mp, lhs, Λ);
  }
  else {
    announce_bad_equation(mp, lhs);
  }
  break;
case mp_vacuous: announce_bad_equation(mp, lhs);
  break;
default:       /* there are no other valid cases, but please the compiler */
  announce_bad_equation(mp, lhs);
  break;
}
check_arith();
mp_recycle_value(mp, lhs);
free_number(v);
```

$mp\_free\_value\_node(mp, lhs)$;
}

1042.   The first argument to try_eq is the location of a value node in a capsule that will soon be recycled. The second argument is either a location within a pair or transform node pointed to by cur_exp, or it is Λ (which means that cur_exp itself serves as the second argument). The idea is to leave cur_exp unchanged, but to equate the two operands.

⟨ Declarations 8 ⟩ +≡
  static void $mp\_try\_eq(\textbf{MP}\ mp, \textbf{mp\_node}\ l, \textbf{mp\_node}\ r)$;

1043.

#define *equation_threshold_k* ((**math_data** ∗) *mp→math*)→*equation_threshold_t*

  static void *deal_with_redundant_or_inconsistent_equation*(**MP** *mp*, **mp_value_node** *p*, **mp_node** *r*)

  {

    mp_number *absp*;

    new_number(*absp*);

    number_clone(*absp*, *value_number*(*p*));

    number_abs(*absp*);

    if (*number_greater*(*absp*, *equation_threshold_k*)) {   /∗ off by .001 or more ∗/

      char *msg*[256];

      const char ∗*hlp*[ ] = {"The␣equation␣I␣just␣read␣contradicts␣what␣was␣said␣before.",

        "But␣don't␣worry;␣continue␣and␣I'll␣just␣ignore␣it.", Λ};

      mp_snprintf(*msg*, 256, "Inconsistent␣equation␣(off␣by␣%s)", *number_tostring*(*value_number*(*p*)));

      mp_back_error(*mp*, *msg*, *hlp*, *true*);

      mp_get_x_next(*mp*);

    }

    else if (*r* ≡ Λ) {

      exclaim_redundant_equation(*mp*);

    }

    free_number(*absp*);

    mp_free_dep_node(*mp*, *p*);

  }

  void *mp_try_eq*(**MP** *mp*, **mp_node** *l*, **mp_node** *r*)

  {

    mp_value_node *p*;   /∗ dependency list for right operand minus left operand ∗/

    mp_variable_type t;   /∗ the type of list *p* ∗/

    mp_value_node *q*;   /∗ the constant term of *p* is here ∗/

    mp_value_node *pp*;   /∗ dependency list for right operand ∗/

    mp_variable_type tt;   /∗ the type of list *pp* ∗/

    boolean *copied*;   /∗ have we copied a list that ought to be recycled? ∗/   /∗ Remove the left

      operand from its container, negate it, and put it into dependency list *p* with constant term *q* ∗/

    t = *mp_type*(*l*);

    if (*t* ≡ *mp_known*) {

      mp_number *arg1*;

      new_number(*arg1*);

      number_clone(*arg1*, *value_number*(*l*));

      number_negate(*arg1*);

      t = *mp_dependent*;

      p = *mp_const_dependency*(*mp*, *arg1*);

      q = *p*;

      free_number(*arg1*);

    }

    else if (*t* ≡ *mp_independent*) {

      t = *mp_dependent*;

      p = *mp_single_dependency*(*mp*, *l*);

      number_negate(*dep_value*(*p*));

      q = *mp→dep_final*;

    }

    else {

```
      mp_value_node ll = (mp_value_node) l;
    p = (mp_value_node) dep_list(ll);
    q = p;
    while (1) {
      number_negate(dep_value(q));
      if (dep_info(q) ≡ Λ) break;
      q = (mp_value_node) mp_link(q);
    }
    mp_link(prev_dep(ll)) = mp_link(q);
    set_prev_dep((mp_value_node) mp_link(q), prev_dep(ll));
    mp_type(ll) = mp_known;
  }     /* Add the right operand to list p */
  if (r ≡ Λ) {
    if (mp→cur_exp.type ≡ mp_known) {
      number_add(value_number(q), cur_exp_value_number());
      goto DONE1;
    }
    else {
      tt = mp→cur_exp.type;
      if (tt ≡ mp_independent)  pp = mp_single_dependency(mp, cur_exp_node());
      else  pp = (mp_value_node) dep_list((mp_value_node) cur_exp_node());
    }
  }
  else {
    if (mp_type(r) ≡ mp_known) {
      number_add(dep_value(q), value_number(r));
      goto DONE1;
    }
    else {
      tt = mp_type(r);
      if (tt ≡ mp_independent)  pp = mp_single_dependency(mp, r);
      else  pp = (mp_value_node) dep_list((mp_value_node) r);
    }
  }
  if (tt ≠ mp_independent) {
    copied = false;
  }
  else {
    copied = true;
    tt = mp_dependent;
  }     /* Add dependency list pp of type tt to dependency list p of type t */
  mp→watch_coefs = false;
  if (t ≡ tt) {
    p = mp_p_plus_q(mp, p, pp, (quarterword) t);
  }
  else if (t ≡ mp_proto_dependent) {
    p = mp_p_plus_fq(mp, p, unity_t, pp, mp_proto_dependent, mp_dependent);
  }
  else {
    mp_number x;
    new_number(x);
    q = p;
```

```
     while (dep_info(q) ≠ Λ) {
        number_clone(x, dep_value(q));
        fraction_to_round_scaled(x);
        set_dep_value(q, x);
        q = (mp_value_node) mp_link(q);
     }
     free_number(x);
     t = mp_proto_dependent;
     p = mp_p_plus_q(mp, p, pp, (quarterword) t);
  }
  mp→watch_coefs = true;
  if (copied) mp_flush_node_list(mp, (mp_node) pp);
DONE1:
  if (dep_info(p) ≡ Λ) {
     deal_with_redundant_or_inconsistent_equation(mp, p, r);
  }
  else {
     mp_linear_eq(mp, p, (quarterword) t);
     if (r ≡ Λ ∧ mp→cur_exp.type ≠ mp_known) {
        if (mp_type(cur_exp_node()) ≡ mp_known) {
           mp_node pp = cur_exp_node();

           set_cur_exp_value_number(value_number(pp));
           mp→cur_exp.type = mp_known;
           mp_free_value_node(mp, pp);
        }
     }
  }
}
```

1044.   Our next goal is to process type declarations. For this purpose it's convenient to have a procedure that scans a ⟨ declared variable ⟩ and returns the corresponding token list. After the following procedure has acted, the token after the declared variable will have been scanned, so it will appear in *cur_cmd*, *cur_mod*, and *cur_sym*.

⟨ Declarations 8 ⟩ +≡
  static mp_node *mp_scan_declared_variable*(**MP** *mp*);

1045.   **mp_node** *mp_scan_declared_variable*(**MP** *mp*)
{
   mp_sym *x*;   /∗ hash address of the variable's root ∗/
   mp_node *h*, *t*;   /∗ head and tail of the token list to be returned ∗/
   mp_get_symbol(*mp*);
   x = *cur_sym*( );
   if (*cur_cmd*( ) ≠ *mp_tag_token*) *mp_clear_symbol*(*mp*, *x*, *false*);
   h = *mp_get_symbolic_node*(*mp*);
   set_mp_sym_sym(*h*, *x*);
   t = *h*;
   while (1) {
     mp_get_x_next(*mp*);
     if (*cur_sym*( ) ≡ Λ) **break**;
     if (*cur_cmd*( ) ≠ *mp_tag_token*) {
       if (*cur_cmd*( ) ≠ *mp_internal_quantity*) {
         if (*cur_cmd*( ) ≡ *mp_left_bracket*) {   /∗ Descend past a collective subscript ∗/
           /∗ If the subscript isn't collective, we don't accept it as part of the declared variable. ∗/
           mp_sym *ll* = *cur_sym*( );   /∗ hash address of left bracket ∗/

           mp_get_x_next(*mp*);
           if (*cur_cmd*( ) ≡ *mp_right_bracket*) {
             set_cur_sym(*collective_subscript*);
           }
           else {
             mp_back_input(*mp*);
             set_cur_sym(*ll*);
             set_cur_cmd((*mp_variable_type*)*mp_left_bracket*);
             break;
           }
         }
         else {
           break;
         }
       }
     }
     mp_link(*t*) = *mp_get_symbolic_node*(*mp*);
     t = *mp_link*(*t*);
     set_mp_sym_sym(*t*, *cur_sym*( ));
     mp_name_type(*t*) = *cur_sym_mod*( );
   }
   if ((*eq_type*(*x*) % *mp_outer_tag*) ≠ *mp_tag_token*) *mp_clear_symbol*(*mp*, *x*, *false*);
   if (*equiv_node*(*x*) ≡ Λ) *mp_new_root*(*mp*, *x*);
   return *h*;
}

1046.   Type declarations are introduced by the following primitive operations.

⟨ Put each of METAPOST's primitives into the hash table 200 ⟩ +≡
  $mp\_primitive(mp, \texttt{"numeric"}, mp\_type\_name, mp\_numeric\_type)$;

  ;

  $mp\_primitive(mp, \texttt{"string"}, mp\_type\_name, mp\_string\_type)$;

  ;

  $mp\_primitive(mp, \texttt{"boolean"}, mp\_type\_name, mp\_boolean\_type)$;

  ;

  $mp\_primitive(mp, \texttt{"path"}, mp\_type\_name, mp\_path\_type)$;

  ;

  $mp\_primitive(mp, \texttt{"pen"}, mp\_type\_name, mp\_pen\_type)$;

  ;

  $mp\_primitive(mp, \texttt{"picture"}, mp\_type\_name, mp\_picture\_type)$;

  ;

  $mp\_primitive(mp, \texttt{"transform"}, mp\_type\_name, mp\_transform\_type)$;

  ;

  $mp\_primitive(mp, \texttt{"color"}, mp\_type\_name, mp\_color\_type)$;

  ;

  $mp\_primitive(mp, \texttt{"rgbcolor"}, mp\_type\_name, mp\_color\_type)$;

  ;

  $mp\_primitive(mp, \texttt{"cmykcolor"}, mp\_type\_name, mp\_cmykcolor\_type)$;

  ;

  $mp\_primitive(mp, \texttt{"pair"}, mp\_type\_name, mp\_pair\_type)$;

1047.   ⟨ Cases of *print_cmd_mod* for symbolic printing of primitives 233 ⟩ +≡
case mp_type_name: $mp\_print\_type(mp, (\textbf{quarterword})\ m)$;
  break;

1048.   Now we are ready to handle type declarations, assuming that a *type_name* has just been scanned.

⟨ Declare action procedures for use by *do_statement* 1048 ⟩ ≡
  static void $mp\_do\_type\_declaration(\textbf{MP}\ mp)$;

See also sections 1074, 1083, 1086, 1091, 1093, 1101, 1103, 1107, 1109, 1111, 1115, 1117, 1119, 1124, 1126, 1131, 1133, 1135, 1137, 1145, 1153, 1177, 1179, 1182, 1244, and 1264.

This code is used in section 1033.

1049.   **static void** *flush_spurious_symbols_after_declared_variable*(**MP** *mp*);

  void *mp_do_type_declaration*(**MP** *mp*)
  {
    integer *t*;   /∗ the type being declared ∗/
    mp_node *p*;   /∗ token list for a declared variable ∗/
    mp_node *q*;   /∗ value node for the variable ∗/
    if (*cur_mod*( ) ≥ *mp_transform_type*) *t* = (**quarterword**) *cur_mod*( );
    else *t* = (**quarterword**)(*cur_mod*( ) + *unknown_tag*);
    do {
      p = *mp_scan_declared_variable*(*mp*);
      mp_flush_variable(*mp*, *equiv_node*(*mp_sym_sym*(*p*)), *mp_link*(*p*), *false*);
      q = *mp_find_variable*(*mp*, *p*);
      if (*q* ≠ Λ) {
        mp_type(*q*) = *t*;
        set_value_number(*q*, *zero_t*);   /∗ todo: this was *null* ∗/
      }
      else {
        const char ∗*hlp*[ ] = {"You␣can't␣use,␣e.g.,␣'numeric␣foo[]'␣after␣'vardef␣foo'.",
            "Proceed,␣and␣I'll␣ignore␣the␣illegal␣redeclaration.", Λ};
        mp_back_error(*mp*, "Declared␣variable␣conflicts␣with␣previous␣vardef", *hlp*, *true*);
        mp_get_x_next(*mp*);
      }
      mp_flush_node_list(*mp*, *p*);
      if (*cur_cmd*( ) < *mp_comma*) {
        flush_spurious_symbols_after_declared_variable(*mp*);
      }
    } **while** (¬*mp_end_of_statement*);
  }

1050.

  static void *flush_spurious_symbols_after_declared_variable*(**MP** *mp*)
  {
    const char ∗*hlp*[ ] = {"Variables␣in␣declarations␣must␣consist␣entirely␣of",
      "names␣and␣collective␣subscripts,␣e.g.,␣'x[]a'.",
      "Are␣you␣trying␣to␣use␣a␣reserved␣word␣in␣a␣variable␣name?",
      "I'm␣going␣to␣discard␣the␣junk␣I␣found␣here,",
      "up␣to␣the␣next␣comma␣or␣the␣end␣of␣the␣declaration.", Λ};
    if (*cur_cmd*( ) ≡ *mp_numeric_token*)
      hlp[2] = "Explicit␣subscripts␣like␣'x15a'␣aren't␣permitted.";
    mp_back_error(*mp*, "Illegal␣suffix␣of␣declared␣variable␣will␣be␣flushed", *hlp*, *true*);
    mp_get_x_next(*mp*);
    mp→*scanner_status* = *flushing*;
    do {
      get_t_next(*mp*);
      ⟨Decrease the string reference count, if the current token is a string 812⟩;
    } **while** (*cur_cmd*( ) < *mp_comma*);   /∗ break on either *end_of_statement* or *comma* ∗/
    mp→*scanner_status* = *normal*;
  }

1051.   METAPOST's *main_control* procedure just calls *do_statement* repeatedly until coming to the end of the user's program. Each execution of *do_statement* concludes with *cur_cmd* = *semicolon*, *end_group*, or *stop*.

```
static void mp_main_control(MP mp)
{
  do {
    mp_do_statement(mp);
    if (cur_cmd() ≡ mp_end_group) {
      mp_value new_expr;
      const char *hlp[] = {"I'm␣not␣currently␣working␣on␣a␣'begingroup',",
          "so␣I␣had␣better␣not␣try␣to␣end␣anything.", Λ};
      memset(&new_expr, 0, sizeof(mp_value));
      new_number(new_expr.data.n);
      mp_error(mp, "Extra␣'endgroup'", hlp, true);
      mp_flush_cur_exp(mp, new_expr);
    }
  } while (cur_cmd() ≠ mp_stop);
}
int mp_run(MP mp)
{
  if (mp→history < mp_fatal_error_stop) {
    xfree(mp→jump_buf);
    mp→jump_buf = malloc(sizeof(jmp_buf));
    if (mp→jump_buf ≡ Λ ∨ setjmp(*(mp→jump_buf)) ≠ 0) return mp→history;
    mp_main_control(mp);      /* come to life */
    mp_final_cleanup(mp);       /* prepare for death */
    mp_close_files_and_terminate(mp);
  }
  return mp→history;
}
```

1052.   This function allows setting of internals from an external source (like the command line or a controlling application).

It accepts two **char** $*$'s, even for numeric assignments when it calls *atoi* to get an integer from the start of the string.

```
void mp_set_internal(MP mp, char *n, char *v, int isstring)
{
    size_t l = strlen(n);
    char err[256];
    const char *errid = Λ;
    if (l > 0) {
        mp_sym p = mp_id_lookup(mp, n, l, false);
        if (p ≡ Λ) {
            errid = "variable␣does␣not␣exist";
        }
        else {
            if (eq_type(p) ≡ mp_internal_quantity) {
                if ((internal_type(equiv(p)) ≡ mp_string_type) ∧ (isstring)) {
                    set_internal_string(equiv(p), mp_rts(mp, v));
                }
                else if ((internal_type(equiv(p)) ≡ mp_known) ∧ (¬isstring)) {
                    int test = atoi(v);
                    if (test > 16383) {
                        errid = "value␣is␣too␣large";
                    }
                    else if (test < −16383) {
                        errid = "value␣is␣too␣small";
                    }
                    else {
                        set_internal_from_number(equiv(p), unity_t);
                        number_multiply_int(internal_value(equiv(p)), test);
                    }
                }
                else {
                    errid = "value␣has␣the␣wrong␣type";
                }
            }
            else {
                errid = "variable␣is␣not␣an␣internal";
            }
        }
    }
    if (errid ≠ Λ) {
        if (isstring) {
            mp_snprintf(err, 256, "%s=\"%s\":␣%s,␣assignment␣ignored.", n, v, errid);
        }
        else {
            mp_snprintf(err, 256, "%s=%d:␣%s,␣assignment␣ignored.", n, atoi(v), errid);
        }
        mp_warn(mp, err);
    }
}
```

1053. ⟨Exported function headers 18⟩ +≡
  void *mp_set_internal*(**MP** *mp*, **char** *∗n*, **char** *∗v*, **int** *isstring*);

1054.   For *mp_execute*, we need to define a structure to store the redirected input and output. This structure holds the five relevant streams: the three informational output streams, the PostScript generation stream, and the input stream. These streams have many things in common, so it makes sense to give them their own structure definition.

 fptr is a virtual file pointer
data is the data this stream holds
  cur is a cursor pointing into *data*
 size is the allocated length of the data stream
used is the actual length of the data stream

   There are small differences between input and output: *term_in* never uses *used*, whereas the other four never use *cur*.

   The file *luatexdir/tex/texfileio.h* defines *term_in* as *stdin* and *term_out* as *stdout*. Moreover *stdio.h* for MinGW defines *stdin* as (&_iob[0]) and *stdout* as (&_iob[1]). We must avoid all that.

⟨Exported types 15⟩ +≡
#undef *term_in*
#undef *term_out*
  typedef struct {
    void *∗fptr*;
    char *∗data*;
    char *∗cur*;
    size_t *size*;
    size_t *used*;
  } **mp_stream**;
  typedef struct {
    mp_stream *term_out*;
    mp_stream *error_out*;
    mp_stream *log_out*;
    mp_stream *ship_out*;
    mp_stream *term_in*;
    struct *mp_edge_object ∗edges*;
  } **mp_run_data**;

1055.   We need a function to clear an output stream, this is called at the beginning of *mp_execute*. We also need one for destroying an output stream, this is called just before a stream is (re)opened.

  static void *mp_reset_stream*(**mp_stream** *∗str*)
  {
    xfree(*str→data*);
    str→cur = Λ;
    str→size = 0;
    str→used = 0;
  }

  static void *mp_free_stream*(**mp_stream** *∗str*)
  {
    xfree(*str→fptr*);
    mp_reset_stream(*str*);
  }

1056.   ⟨ Declarations 8 ⟩ +≡
  static void *mp_reset_stream*(**mp_stream** *∗str*);
  static void *mp_free_stream*(**mp_stream** *∗str*);

1057.   The global instance contains a pointer instead of the actual structure even though it is essentially static, because that makes it is easier to move the object around.

⟨ Global variables 14 ⟩ +≡
  mp_run_data *run_data*;

1058.   Another type is needed: the indirection will overload some of the file pointer objects in the instance (but not all). For clarity, an indirect object is used that wraps a **FILE** ∗.

⟨ Types in the outer block 33 ⟩ +≡
  typedef struct File {
    FILE *∗f*;
  } **File**;

1059.   Here are all of the functions that need to be overloaded for *mp_execute*.

⟨ Declarations 8 ⟩ +≡
  static void *∗mplib_open_file*(**MP** *mp*, **const char** *∗fname*, **const char** *∗fmode*, **int** *ftype*);
  static int *mplib_get_char*(**void** *∗f*, **mp_run_data** *∗mplib_data*);
  static void *mplib_unget_char*(**void** *∗f*, **mp_run_data** *∗mplib_data*, **int** *c*);
  static char *∗mplib_read_ascii_file*(**MP** *mp*, **void** *∗ff*, **size_t** *∗size*);
  static void *mplib_write_ascii_file*(**MP** *mp*, **void** *∗ff*, **const char** *∗s*);
  static void *mplib_read_binary_file*(**MP** *mp*, **void** *∗ff*, **void** *∗∗data*, **size_t** *∗size*);
  static void *mplib_write_binary_file*(**MP** *mp*, **void** *∗ff*, **void** *∗s*, **size_t** *size*);
  static void *mplib_close_file*(**MP** *mp*, **void** *∗ff*);
  static int *mplib_eof_file*(**MP** *mp*, **void** *∗ff*);
  static void *mplib_flush_file*(**MP** *mp*, **void** *∗ff*);
  static void *mplib_shipout_backend*(**MP** *mp*, **void** *∗h*);

1060.   The $xmalloc(1,1)$ calls make sure the stored indirection values are unique.

#define $reset\_stream(a)$ **do**
         {
             $mp\_reset\_stream(\&(a));$
              if $(\neg ff\!\rightarrow\!\!f)$ {
                  $ff\!\rightarrow\!\!f = xmalloc(1,1);$
                  $(a).fptr = ff\!\rightarrow\!\!f;$
              }
         }
         while (0)

  static void $*mplib\_open\_file(\mathbf{MP}\ mp, \mathbf{const\ char}\ *fname, \mathbf{const\ char}\ *fmode, \mathbf{int}\ ftype)$
  {
    File $*ff = xmalloc(1, \mathbf{sizeof}(\mathbf{File}));$
    mp_run_data $*run = mp\_rundata(mp);$

    $ff\!\rightarrow\!\!f = \Lambda;$
    if $(ftype \equiv mp\_filetype\_terminal)$ {
       if $(fmode[0] \equiv \mathtt{'r'})$ {
          if $(\neg ff\!\rightarrow\!\!f)$ {
              $ff\!\rightarrow\!\!f = xmalloc(1,1);$
              $run\!\rightarrow\!term\_in.fptr = ff\!\rightarrow\!\!f;$
          }
       }
        else {
           $reset\_stream(run\!\rightarrow\!term\_out);$
        }
    }
    else if $(ftype \equiv mp\_filetype\_error)$ {
       $reset\_stream(run\!\rightarrow\!error\_out);$
    }
    else if $(ftype \equiv mp\_filetype\_log)$ {
       $reset\_stream(run\!\rightarrow\!log\_out);$
    }
    else if $(ftype \equiv mp\_filetype\_postscript)$ {
       $mp\_free\_stream(\&(run\!\rightarrow\!ship\_out));$
       $ff\!\rightarrow\!\!f = xmalloc(1,1);$
       $run\!\rightarrow\!ship\_out.fptr = ff\!\rightarrow\!\!f;$
    }
    else if $(ftype \equiv mp\_filetype\_bitmap)$ {
       $mp\_free\_stream(\&(run\!\rightarrow\!ship\_out));$
       $ff\!\rightarrow\!\!f = xmalloc(1,1);$
       $run\!\rightarrow\!ship\_out.fptr = ff\!\rightarrow\!\!f;$
    }
    else {
       char $realmode[3];$
       char $*f = (mp\!\rightarrow\!find\_file)(mp, fname, fmode, ftype);$

       if $(f \equiv \Lambda)$ **return** $\Lambda;$
       $realmode[0] = *fmode;$
       $realmode[1] = \mathtt{'b'};$
       $realmode[2] = 0;$
       $ff\!\rightarrow\!\!f = fopen(f, realmode);$
       $free(f);$

```
      if ((fmode[0] ≡ 'r') ∧ (ff→f ≡ Λ)) {
        free(ff);
        return Λ;
      }
    }
  }
  return ff;
}
static int mplib_get_char(void *f, mp_run_data *run)
{
  int c;
  if (f ≡ run→term_in.fptr ∧ run→term_in.data ≠ Λ) {
    if (run→term_in.size ≡ 0) {
      if (run→term_in.cur ≠ Λ) {
        run→term_in.cur = Λ;
      }
      else {
        xfree(run→term_in.data);
      }
      c = EOF;
    }
    else {
      run→term_in.size −−;
      c = *(run→term_in.cur)++;
    }
  }
  else {
    c = fgetc(f);
  }
  return c;
}
static void mplib_unget_char(void *f, mp_run_data *run, int c)
{
  if (f ≡ run→term_in.fptr ∧ run→term_in.cur ≠ Λ) {
    run→term_in.size ++;
    run→term_in.cur −−;
  }
  else {
    ungetc(c, f);
  }
}
static char *mplib_read_ascii_file(MP mp, void *ff, size_t *size)
{
  char *s = Λ;
  if (ff ≠ Λ) {
    int c;
    size_t len = 0, lim = 128;
    mp_run_data *run = mp_rundata(mp);
    FILE *f = ((File *) ff)→f;
    if (f ≡ Λ) return Λ;
    *size = 0;
    c = mplib_get_char(f, run);
```

```
    if (c ≡ EOF) return Λ;
    s = malloc(lim);
    if (s ≡ Λ) return Λ;
    while (c ≠ EOF ∧ c ≠ '\n' ∧ c ≠ '\r') {
      if (len ≥ (lim − 1)) {
        s = xrealloc(s, (lim + (lim ≫ 2)), 1);
        if (s ≡ Λ) return Λ;
        lim += (lim ≫ 2);
      }
      s[len ++] = (char) c;
      c = mplib_get_char(f, run);
    }
    if (c ≡ '\r') {
      c = mplib_get_char(f, run);
      if (c ≠ EOF ∧ c ≠ '\n') mplib_unget_char(f, run, c);
    }
    s[len] = 0;
    ∗size = len;
  }
  return s;
}
static void mp_append_string(MP mp, mp_stream ∗a, const char ∗b)
{
  size_t l = strlen(b) + 1;      /∗ don't forget the trailing '\0' ∗/
  if ((a⃗used + l) ≥ a⃗size) {
    a⃗size += 256 + (a⃗size)/5 + l;
    a⃗data = xrealloc(a⃗data, a⃗size, 1);
  }
  memcpy(a⃗data + a⃗used, b, l);
  a⃗used += (l − 1);
}
static void mp_append_data(MP mp, mp_stream ∗a, void ∗b, size_t l)
{
  if ((a⃗used + l) ≥ a⃗size) {
    a⃗size += 256 + (a⃗size)/5 + l;
    a⃗data = xrealloc(a⃗data, a⃗size, 1);
  }
  memcpy(a⃗data + a⃗used, b, l);
  a⃗used += l;
}
static void mplib_write_ascii_file(MP mp, void ∗ff, const char ∗s)
{
  if (ff ≠ Λ) {
    void ∗f = ((File ∗) ff)⃗f;
    mp_run_data ∗run = mp_rundata(mp);
    if (f ≠ Λ) {
      if (f ≡ run⃗term_out.fptr) {
        mp_append_string(mp, &(run⃗term_out), s);
      }
      else if (f ≡ run⃗error_out.fptr) {
        mp_append_string(mp, &(run⃗error_out), s);
```

```
      }
      else if (f ≡ run→log_out.fptr) {
        mp_append_string(mp, &(run→log_out), s);
      }
      else if (f ≡ run→ship_out.fptr) {
        mp_append_string(mp, &(run→ship_out), s);
      }
      else {
        fprintf((FILE *) f, "%s", s);
      }
    }
  }
}
static void mplib_read_binary_file(MP mp, void *ff, void **data, size_t *size)
{
  (void) mp;
  if (ff ≠ Λ) {
    size_t len = 0;
    FILE *f = ((File *) ff)→f;
    if (f ≠ Λ) len = fread(*data, 1, *size, f);
    *size = len;
  }
}
static void mplib_write_binary_file(MP mp, void *ff, void *s, size_t size)
{
  (void) mp;
  if (ff ≠ Λ) {
    void *f = ((File *) ff)→f;
    mp_run_data *run = mp_rundata(mp);
    if (f ≠ Λ) {
      if (f ≡ run→ship_out.fptr) {
        mp_append_data(mp, &(run→ship_out), s, size);
      }
      else {
        (void) fwrite(s, size, 1, f);
      }
    }
  }
}
static void mplib_close_file(MP mp, void *ff)
{
  if (ff ≠ Λ) {
    mp_run_data *run = mp_rundata(mp);
    void *f = ((File *) ff)→f;
    if (f ≠ Λ) {
      if (f ≠ run→term_out.fptr ∧ f ≠ run→error_out.fptr ∧ f ≠ run→log_out.fptr ∧ f ≠
          run→ship_out.fptr ∧ f ≠ run→term_in.fptr) {
        fclose(f);
      }
    }
    free(ff);
```

```
    }
  }
  static int mplib_eof_file(MP mp, void *ff )
  {
    if (ff ≠ Λ) {
      mp_run_data *run = mp_rundata(mp);
      FILE *f = ((File *) ff )→f;
      if (f ≡ Λ) return 1;
      if (f ≡ run→term_in.fptr ∧ run→term_in.data ≠ Λ) {
        return (run→term_in.size ≡ 0);
      }
      return feof (f);
    }
    return 1;
  }
  static void mplib_flush_file(MP mp, void *ff )
  {
    (void) mp;
    (void) ff ;
    return;
  }
  static void mplib_shipout_backend(MP mp, void *voidh)
  {
    mp_edge_header_node h = (mp_edge_header_node) voidh;
    mp_edge_object * hh = mp_gr_export(mp, h);
    if (hh) {
      mp_run_data *run = mp_rundata(mp);
      if (run→edges ≡ Λ) {
        run→edges = hh;
      }
      else {
        mp_edge_object * p = run→edges;
        while (p→next ≠ Λ) {
          p = p→next;
        }
        p→next = hh;
      }
    }
  }
```

1061.   This is where we fill them all in.

⟨ Prepare function pointers for non-interactive use 1061 ⟩ ≡

  {

    $mp\rightarrow open\_file = mplib\_open\_file$;

    $mp\rightarrow close\_file = mplib\_close\_file$;

    $mp\rightarrow eof\_file = mplib\_eof\_file$;

    $mp\rightarrow flush\_file = mplib\_flush\_file$;

    $mp\rightarrow write\_ascii\_file = mplib\_write\_ascii\_file$;

    $mp\rightarrow read\_ascii\_file = mplib\_read\_ascii\_file$;

    $mp\rightarrow write\_binary\_file = mplib\_write\_binary\_file$;

    $mp\rightarrow read\_binary\_file = mplib\_read\_binary\_file$;

    $mp\rightarrow shipout\_backend = mplib\_shipout\_backend$;

  }

This code is used in section 16.

1062.   Perhaps this is the most important API function in the library.

⟨ Exported function headers 18 ⟩ +≡

  extern mp_run_data *$mp\_rundata$(**MP** mp);

1063.   **mp_run_data** *$mp\_rundata$(**MP** mp)

  {

    return &($mp\rightarrow run\_data$);

  }

1064.   ⟨ Dealloc variables 27 ⟩ +≡

  mp_free_stream(&($mp\rightarrow run\_data.term\_in$));

  mp_free_stream(&($mp\rightarrow run\_data.term\_out$));

  mp_free_stream(&($mp\rightarrow run\_data.log\_out$));

  mp_free_stream(&($mp\rightarrow run\_data.error\_out$));

  mp_free_stream(&($mp\rightarrow run\_data.ship\_out$));

1065.   ⟨ Finish non-interactive use 1065 ⟩ ≡

  $xfree$($mp\rightarrow term\_out$);

  $xfree$($mp\rightarrow term\_in$);

  $xfree$($mp\rightarrow err\_out$);

This code is used in section 12.

1066.   ⟨Start non-interactive work 1066⟩ ≡

⟨Initialize the output routines 81⟩;

mp→$input\_ptr = 0$;

mp→$max\_in\_stack = file\_bottom$;

mp→$in\_open = file\_bottom$;

mp→$open\_parens = 0$;

mp→$max\_buf\_stack = 0$;

mp→$param\_ptr = 0$;

mp→$max\_param\_stack = 0$;

$start = loc = 0$;

$iindex = file\_bottom$;

$nloc = nstart = \Lambda$;

mp→$first = 0$; **line** $= 0$;

$name = is\_term$;

mp→$mpx\_name[file\_bottom] = absent$;

mp→$force\_eof = false$;

$t\_open\_in( )$;

mp→$scanner\_status = normal$;

if $(\neg mp$→$ini\_version)$ {

  if $(\neg mp\_load\_preload\_file(mp))$ {

    mp→$history = mp\_fatal\_error\_stop$;

    return mp→$history$;

  }

}

$mp\_fix\_date\_and\_time(mp)$;

if $(mp$→$random\_seed \equiv 0)$

  mp→$random\_seed = (number\_to\_scaled(internal\_value(mp\_time))/number\_to\_scaled(unity\_t)) +$

    $number\_to\_scaled(internal\_value(mp\_day))$;

$init\_randoms(mp$→$random\_seed)$;

$initialize\_print\_selector( )$;

$mp\_open\_log\_file(mp)$;

$mp\_set\_job\_id(mp)$;

$mp\_init\_map\_file(mp, mp$→$troff\_mode)$;

mp→$history = mp\_spotless$;   /∗ ready to go! ∗/

if $(mp$→$troff\_mode)$ {

  $number\_clone(internal\_value(mp\_gtroffmode), unity\_t)$;

  $number\_clone(internal\_value(mp\_prologues), unity\_t)$;

}

⟨Fix up mp→$internal[mp\_job\_name]$ 868⟩;

if $(mp$→$start\_sym \neq \Lambda)$ {   /∗ insert the '**everyjob**' symbol ∗/

  $set\_cur\_sym(mp$→$start\_sym)$;

  $mp\_back\_input(mp)$;

}

This code is used in section 1067.

1067.   **int** *mp_execute*(**MP** *mp*, **char** *∗s*, **size_t** *l*)
  {
    mp_reset_stream(&(*mp⃗run_data.term_out*));
    mp_reset_stream(&(*mp⃗run_data.log_out*));
    mp_reset_stream(&(*mp⃗run_data.error_out*));
    mp_reset_stream(&(*mp⃗run_data.ship_out*));
    if (*mp⃗finished*) {
      return *mp⃗history*;
    }
    else if (¬*mp⃗noninteractive*) {
      mp⃗history = *mp_fatal_error_stop*;
      return *mp⃗history*;
    }
    if (*mp⃗history* < *mp_fatal_error_stop*) {
      xfree(*mp⃗jump_buf*);
      mp⃗jump_buf = *malloc*(**sizeof**(**jmp_buf**));
      if (*mp⃗jump_buf* ≡ Λ ∨ *setjmp*(∗(*mp⃗jump_buf*)) ≠ 0) {
        return *mp⃗history*;
      }
      if (*s* ≡ Λ) {   /∗ this signals EOF ∗/
        mp_final_cleanup(*mp*);   /∗ prepare for death ∗/
        mp_close_files_and_terminate(*mp*);
        return *mp⃗history*;
      }
      mp⃗tally = 0;
      mp⃗term_offset = 0;
      mp⃗file_offset = 0;   /∗ Perhaps some sort of warning here when *data* is not ∗ yet exhausted would
          be nice ... this happens after errors ∗/
      if (*mp⃗run_data.term_in.data*) *xfree*(*mp⃗run_data.term_in.data*);
      mp⃗run_data.term_in.data = *xstrdup*(*s*);
      mp⃗run_data.term_in.cur = *mp⃗run_data.term_in.data*;
      mp⃗run_data.term_in.size = *l*;
      if (*mp⃗run_state* ≡ 0) {
        mp⃗selector = *term_only*;
        ⟨ Start non-interactive work 1066 ⟩;
      }
      mp⃗run_state = 1;
      (**void**) *mp_input_ln*(*mp*, *mp⃗term_in*);
      mp_firm_up_the_line(*mp*);
      mp⃗buffer[*limit*] = *xord*('%');
      mp⃗first = (**size_t**)(*limit* + 1);
      loc = *start*;
      do {
        mp_do_statement(*mp*);
      } **while** (*cur_cmd*( ) ≠ *mp_stop*);
      mp_final_cleanup(*mp*);
      mp_close_files_and_terminate(*mp*);
    }
    return *mp⃗history*;
  }

1068.   This function cleans up

```
int mp_finish(MP mp)
{
    int history = 0;
    if (mp→finished ∨ mp→history ≥ mp_fatal_error_stop) {
        history = mp→history;
        mp_free(mp);
        return history;
    }
    xfree(mp→jump_buf);
    mp→jump_buf = malloc(sizeof(jmp_buf));
    if (mp→jump_buf ≡ Λ ∨ setjmp(*(mp→jump_buf)) ≠ 0) {
        history = mp→history;
    }
    else {
        history = mp→history;
        mp_final_cleanup(mp);       /* prepare for death */
    }
    mp_close_files_and_terminate(mp);
    mp_free(mp);
    return history;
}
```

1069.   People may want to know the library version

```
char *mp_metapost_version(void)
{
    return mp_strdup(metapost_version);
}
void mp_show_library_versions(void)
{
    fprintf(stdout, "Compiled␣with␣cairo␣%s;␣using␣%s\n", CAIRO_VERSION_STRING,
        cairo_version_string());
    fprintf(stdout, "Compiled␣with␣pixman␣%s;␣using␣%s\n", PIXMAN_VERSION_STRING,
        pixman_version_string());
    fprintf(stdout, "Compiled␣with␣libpng␣%s;␣using␣%s\n", PNG_LIBPNG_VER_STRING, png_libpng_ver);
    fprintf(stdout, "Compiled␣with␣zlib␣%s;␣using␣%s\n", ZLIB_VERSION, zlibVersion());
    fprintf(stdout, "Compiled␣with␣mpfr␣%s;␣using␣%s\n", MPFR_VERSION_STRING, mpfr_get_version());
    fprintf(stdout, "Compiled␣with␣gmp␣%d.%d.%d;␣using␣%s\n\n", __GNU_MP_VERSION,
        __GNU_MP_VERSION_MINOR, __GNU_MP_VERSION_PATCHLEVEL, gmp_version);
}
```

1070.   ⟨Exported function headers 18⟩ +≡

```
int mp_run(MP mp);
int mp_execute(MP mp, char *s, size_t l);
int mp_finish(MP mp);
char *mp_metapost_version(void);
void mp_show_library_versions(void);
```

1071.   ⟨Put each of METAPOST's primitives into the hash table 200⟩ +≡
  $mp\_primitive(mp,\texttt{"end"},mp\_stop,0);$
  ;
  $mp\_primitive(mp,\texttt{"dump"},mp\_stop,1);$
  $mp\text{-}frozen\_dump = mp\_frozen\_primitive(mp,\texttt{"dump"},mp\_stop,1);$

1072.   ⟨Cases of *print_cmd_mod* for symbolic printing of primitives 233⟩ +≡
case mp_stop:
  if $(cur\_mod() \equiv 0)$ $mp\_print(mp,\texttt{"end"});$
  else $mp\_print(mp,\texttt{"dump"});$
  break;

1073.   Commands.   Let's turn now to statements that are classified as "commands" because of their imperative nature. We'll begin with simple ones, so that it will be clear how to hook command processing into the *do_statement* routine; then we'll tackle the tougher commands.

Here's one of the simplest:

1074.   ⟨Declare action procedures for use by *do_statement* 1048⟩ +≡
  static void *mp_do_random_seed*(**MP** *mp*);

1075.   **void** *mp_do_random_seed*(**MP** *mp*)
  {
    mp_value *new_expr*;

    memset(&*new_expr*, 0, **sizeof**(**mp_value**));
    new_number(*new_expr*.*data*.*n*);
    mp_get_x_next(*mp*);
    if (*cur_cmd*() ≠ *mp_assignment*) {
      const char ∗*hlp*[ ] = {"Always␣say␣'randomseed:=<numeric␣expression>'.", Λ};

      mp_back_error(*mp*, "Missing␣':='␣has␣been␣inserted", *hlp*, *true*);
      ;
    }
    ;
    mp_get_x_next(*mp*);
    mp_scan_expression(*mp*);
    if (*mp*→*cur_exp*.*type* ≠ *mp_known*) {
      const char ∗*hlp*[ ] = {"Your␣expression␣was␣too␣random␣for␣me␣to␣handle,",
          "so␣I␣won't␣change␣the␣random␣seed␣just␣now.", Λ};

      mp_disp_err(*mp*, Λ);
      mp_back_error(*mp*, "Unknown␣value␣will␣be␣ignored", *hlp*, *true*);
      ;
      mp_get_x_next(*mp*);
      mp_flush_cur_exp(*mp*, *new_expr*);
    }
    else {
      ⟨Initialize the random seed to *cur_exp* 1076⟩;
    }
  }

1076.   ⟨Initialize the random seed to *cur_exp* 1076⟩ ≡
  {
    init_randoms(*number_to_scaled*(*cur_exp_value_number*()));
    if (*mp*→*selector* ≥ *log_only* ∧ *mp*→*selector* < *write_file*) {
      mp→*old_setting* = *mp*→*selector*;
      mp→*selector* = *log_only*;
      mp_print_nl(*mp*, "{randomseed:=");
      print_number(*cur_exp_value_number*());
      mp_print_char(*mp*, *xord*('}'));
      mp_print_nl(*mp*, "");
      mp→*selector* = *mp*→*old_setting*;
    }
  }

This code is used in section 1075.

1077.   And here's another simple one (somewhat different in flavor):

1078.   ⟨Put each of METAPOST's primitives into the hash table 200⟩ +≡
  $mp\_primitive(mp,$ "batchmode"$, mp\_mode\_command, mp\_batch\_mode);$
  ;
  $mp\_primitive(mp,$ "nonstopmode"$, mp\_mode\_command, mp\_nonstop\_mode);$
  ;
  $mp\_primitive(mp,$ "scrollmode"$, mp\_mode\_command, mp\_scroll\_mode);$
  ;
  $mp\_primitive(mp,$ "errorstopmode"$, mp\_mode\_command, mp\_error\_stop\_mode);$

1079.   ⟨Cases of *print_cmd_mod* for symbolic printing of primitives 233⟩ +≡
case $mp\_mode\_command$:
  switch (m) {
  case mp_batch_mode: $mp\_print(mp,$ "batchmode"$);$
    break;
  case $mp\_nonstop\_mode$: $mp\_print(mp,$ "nonstopmode"$);$
    break;
  case mp_scroll_mode: $mp\_print(mp,$ "scrollmode"$);$
    break;
  default: $mp\_print(mp,$ "errorstopmode"$);$
    break;
  }
  break;

1080.   The '**inner**' and '**outer**' commands are only slightly harder.

1081.   ⟨Put each of METAPOST's primitives into the hash table 200⟩ +≡
  $mp\_primitive(mp,$ "inner"$, mp\_protection\_command, 0);$
  ;
  $mp\_primitive(mp,$ "outer"$, mp\_protection\_command, 1);$

1082.   ⟨Cases of *print_cmd_mod* for symbolic printing of primitives 233⟩ +≡
case $mp\_protection\_command$:
  if $(m \equiv 0)$ $mp\_print(mp,$ "inner"$);$
  else $mp\_print(mp,$ "outer"$);$
  break;

1083.   ⟨Declare action procedures for use by *do_statement* 1048⟩ +≡
  static void $mp\_do\_protection(\textbf{MP}\ mp);$

1084.   **void** $mp\_do\_protection(\textbf{MP }mp)$
{
    int m;   /∗ 0 to unprotect, 1 to protect ∗/
    halfword t;   /∗ the eq_type before we change it ∗/
    $m = cur\_mod(\ )$;
    do {
      $mp\_get\_symbol(mp)$;
      $t = eq\_type(cur\_sym(\ ))$;
      if $(m \equiv 0)$ {
        if $(t \geq mp\_outer\_tag)$ $set\_eq\_type(cur\_sym(\ ), (t - mp\_outer\_tag))$;
      }
      else if $(t < mp\_outer\_tag)$ {
        $set\_eq\_type(cur\_sym(\ ), (t + mp\_outer\_tag))$;
      }
      $mp\_get\_x\_next(mp)$;
    } **while** $(cur\_cmd(\ ) \equiv mp\_comma)$;
}

1085.   METAPOST never defines the tokens '(' and ')' to be primitives, but plain METAPOST begins with the declaration '**delimiters** ()'. Such a declaration assigns the command code *left_delimiter* to '(' and *right_delimiter* to ')'; the *equiv* of each delimiter is the hash address of its mate.

1086.   ⟨ Declare action procedures for use by *do_statement* 1048 ⟩ +≡
  static void $mp\_def\_delims(\textbf{MP }mp)$;

1087.   **void** $mp\_def\_delims(\textbf{MP }mp)$
{
    mp_sym $l\_delim,\ r\_delim$;   /∗ the new delimiter pair ∗/
    $mp\_get\_clear\_symbol(mp)$;
    $l\_delim = cur\_sym(\ )$;
    $mp\_get\_clear\_symbol(mp)$;
    $r\_delim = cur\_sym(\ )$;
    $set\_eq\_type(l\_delim, mp\_left\_delimiter)$;
    $set\_equiv\_sym(l\_delim, r\_delim)$;
    $set\_eq\_type(r\_delim, mp\_right\_delimiter)$;
    $set\_equiv\_sym(r\_delim, l\_delim)$;
    $mp\_get\_x\_next(mp)$;
}

1088.   Here is a procedure that is called when METAPOST has reached a point where some right delimiter is mandatory.

⟨ Declarations 8 ⟩ +≡
  static void $mp\_check\_delimiter(\textbf{MP }mp, \textbf{mp\_sym }l\_delim, \textbf{mp\_sym }r\_delim)$;

1089.   **void** *mp_check_delimiter* (**MP** *mp*, **mp_sym** *l_delim*, **mp_sym** *r_delim*)
  {
    if (*cur_cmd* ( ) ≡ *mp_right_delimiter*)
      if (*equiv_sym* (*cur_sym* ( )) ≡ *l_delim*) **return**;
    if (*cur_sym* ( ) ≠ *r_delim*) {
      char *msg* [256];
      const char *∗hlp* [ ] = {"I␣found␣no␣right␣delimiter␣to␣match␣a␣left␣one.␣So␣I've",
          "put␣one␣in,␣behind␣the␣scenes;␣this␣may␣fix␣the␣problem.", Λ};
      mp_snprintf (*msg*, 256, "Missing␣'%s'␣has␣been␣inserted", *mp_str* (*mp*, *text* (*r_delim*)));
      ;
      mp_back_error (*mp*, *msg*, *hlp*, *true*);
    }
    else {
      char *msg* [256];
      const char *∗hlp* [ ] = {"Strange:␣This␣token␣has␣lost␣its␣former␣meaning!",
          "I'll␣read␣it␣as␣a␣right␣delimiter␣this␣time;",
          "but␣watch␣out,␣I'll␣probably␣miss␣it␣later.", Λ};
      mp_snprintf (*msg*, 256, "The␣token␣'%s'␣is␣no␣longer␣a␣right␣delimiter", *mp_str* (*mp*,
          text (*r_delim*)));
      ;
      mp_error (*mp*, *msg*, *hlp*, *true*);
    }
  }

1090.   The next four commands save or change the values associated with tokens.

1091.   ⟨ Declare action procedures for use by *do_statement* 1048 ⟩ +≡
  static void *mp_do_statement* (**MP** *mp*);
  static void *mp_do_interim* (**MP** *mp*);

1092.   **void** *mp_do_interim* (**MP** *mp*)
  {
    mp_get_x_next (*mp*);
    if (*cur_cmd* ( ) ≠ *mp_internal_quantity*) {
      char *msg* [256];
      const char *∗hlp* [ ] = {"Something␣like␣'tracingonline'␣should␣follow␣'interim'.", Λ};
      mp_snprintf (*msg*, 256, "The␣token␣'%s'␣isn't␣an␣internal␣quantity",
          (*cur_sym* ( ) ≡ Λ ? "(%CAPSULE)" : *mp_str* (*mp*, *text* (*cur_sym* ( )))));
      ;
      mp_back_error (*mp*, *msg*, *hlp*, *true*);
    }
    else {
      mp_save_internal (*mp*, *cur_mod* ( ));
      mp_back_input (*mp*);
    }
    mp_do_statement (*mp*);
  }

1093.   The following procedure is careful not to undefine the left-hand symbol too soon, lest commands
like '`let x=x`' have a surprising effect.

⟨ Declare action procedures for use by *do_statement* 1048 ⟩ +≡
  static void *mp_do_let*(**MP** *mp*);

1094.   **void** *mp_do_let*(**MP** *mp*)
  {
    mp_sym *l*;   /∗ hash location of the left-hand symbol ∗/
    mp_get_symbol(*mp*);
    l = *cur_sym*( );
    mp_get_x_next(*mp*);
    if (*cur_cmd*( ) ≠ *mp_equals* ∧ *cur_cmd*( ) ≠ *mp_assignment*) {
      const char ∗*hlp*[ ] = {"You␣should␣have␣said␣'let␣symbol␣=␣something'.",
          "But␣don't␣worry;␣I'll␣pretend␣that␣an␣equals␣sign",
          "was␣present.␣The␣next␣token␣I␣read␣will␣be␣'something'.", Λ};

      mp_back_error(*mp*, "Missing␣'='␣has␣been␣inserted", *hlp*, *true*);
      ;
    }
    mp_get_symbol(*mp*);
    switch (*cur_cmd*( )) {
    case *mp_defined_macro*: **case** *mp_secondary_primary_macro*: **case** *mp_tertiary_secondary_macro*:
      case *mp_expression_tertiary_macro*: *add_mac_ref*(*cur_mod_node*( ));
      break;
    default: **break**;
    }
    mp_clear_symbol(*mp*, *l*, *false*);
    set_eq_type(*l*, *cur_cmd*( ));
    if (*cur_cmd*( ) ≡ *mp_tag_token*) *set_equiv*(*l*, 0);   /∗ todo: this was *null* ∗/
    else if (*cur_cmd*( ) ≡ *mp_defined_macro* ∨ *cur_cmd*( ) ≡ *mp_secondary_primary_macro* ∨ *cur_cmd*( ) ≡
          mp_tertiary_secondary_macro ∨ *cur_cmd*( ) ≡ *mp_expression_tertiary_macro*)
      set_equiv_node(*l*, *cur_mod_node*( ));
    else if (*cur_cmd*( ) ≡ *mp_left_delimiter* ∨ *cur_cmd*( ) ≡ *mp_right_delimiter*)
      set_equiv_sym(*l*, *equiv_sym*(*cur_sym*( )));
    else *set_equiv*(*l*, *cur_mod*( ));
    mp_get_x_next(*mp*);
  }

1095.   ⟨ Declarations 8 ⟩ +≡
  static void *mp_do_new_internal*(**MP** *mp*);

1096.   ⟨ Internal library declarations 10 ⟩ +≡
  void *mp_grow_internals*(**MP** *mp*, **int** *l*);

1097.   **void** $mp\_grow\_internals$(**MP** mp, **int** l)
```
{
  mp_internal *internal;
  int k;
  if (l > max_halfword) {
    mp_confusion(mp, "out␣of␣memory␣space");      /∗ can't be reached ∗/
  }
  internal = xmalloc((l + 1), sizeof(mp_internal));
  for (k = 0; k ≤ l; k++) {
    if (k ≤ mp→max_internal) {
      memcpy(internal + k, mp→internal + k, sizeof(mp_internal));
    }
    else {
      memset(internal + k, 0, sizeof(mp_internal));
      new_number(((mp_internal *)(internal + k))→v.data.n);
    }
  }
  xfree(mp→internal);
  mp→internal = internal;
  mp→max_internal = l;
}
void mp_do_new_internal(MP mp)
{
  int the_type = mp_known;
  mp_get_x_next(mp);
  if (cur_cmd( ) ≡ mp_type_name ∧ cur_mod( ) ≡ mp_string_type) {
    the_type = mp_string_type;
  }
  else {
    if (¬(cur_cmd( ) ≡ mp_type_name ∧ cur_mod( ) ≡ mp_numeric_type)) {
      mp_back_input(mp);
    }
  }
  do {
    if (mp→int_ptr ≡ mp→max_internal) {
      mp_grow_internals(mp, (mp→max_internal + (mp→max_internal/4)));
    }
    mp_get_clear_symbol(mp);
    incr(mp→int_ptr);
    set_eq_type(cur_sym( ), mp_internal_quantity);
    set_equiv(cur_sym( ), mp→int_ptr);
    if (internal_name(mp→int_ptr) ≠ Λ) xfree(internal_name(mp→int_ptr));
    set_internal_name(mp→int_ptr, mp_xstrdup(mp, mp_str(mp, text(cur_sym( )))));
    if (the_type ≡ mp_string_type) {
      set_internal_string(mp→int_ptr, mp_rts(mp, ""));
    }
    else {
      set_number_to_zero(internal_value(mp→int_ptr));
    }
    set_internal_type(mp→int_ptr, the_type);
    mp_get_x_next(mp);
```

```
  } while (cur_cmd( ) ≡ mp_comma);
}
```

1098.   ⟨Dealloc variables 27⟩ +≡
```
  for (k = 0; k ≤ mp→max_internal; k++) {
    free_number(mp→internal[k].v.data.n);
    xfree(internal_name(k));
  }
  xfree(mp→internal);
```

1099.   The various 'show' commands are distinguished by modifier fields in the usual way.

#define *show_token_code* 0   /∗ show the meaning of a single token ∗/
#define *show_stats_code* 1   /∗ show current memory and string usage ∗/
#define *show_code* 2   /∗ show a list of expressions ∗/
#define *show_var_code* 3   /∗ show a variable and its descendents ∗/
#define *show_dependencies_code* 4   /∗ show dependent variables in terms of independents ∗/

⟨Put each of METAPOST's primitives into the hash table 200⟩ +≡
```
  mp_primitive(mp, "showtoken", mp_show_command, show_token_code);
  ;
  mp_primitive(mp, "showstats", mp_show_command, show_stats_code);
  ;
  mp_primitive(mp, "show", mp_show_command, show_code);
  ;
  mp_primitive(mp, "showvariable", mp_show_command, show_var_code);
  ;
  mp_primitive(mp, "showdependencies", mp_show_command, show_dependencies_code);
```

1100.   ⟨Cases of *print_cmd_mod* for symbolic printing of primitives 233⟩ +≡
```
case mp_show_command:
  switch (m) {
  case show_token_code: mp_print(mp, "showtoken");
    break;
  case show_stats_code: mp_print(mp, "showstats");
    break;
  case show_code: mp_print(mp, "show");
    break;
  case show_var_code: mp_print(mp, "showvariable");
    break;
  default: mp_print(mp, "showdependencies");
    break;
  }
  break;
```

1101.   The value of *cur_mod* controls the *verbosity* in the *print_exp* routine: if it's *show_code*, complicated structures are abbreviated, otherwise they aren't.

⟨Declare action procedures for use by *do_statement* 1048⟩ +≡
```
  static void mp_do_show(MP mp);
```

1102.   **void** $mp\_do\_show(\mathbf{MP}\ mp)$
{
  mp_value new_expr;
  do {
    $memset(\&new\_expr, 0, \mathbf{sizeof}(\mathbf{mp\_value}))$;
    $new\_number(new\_expr.data.n)$;
    $mp\_get\_x\_next(mp)$;
    $mp\_scan\_expression(mp)$;
    $mp\_print\_nl(mp, ">>_\sqcup")$;
    ;
    $mp\_print\_exp(mp, \Lambda, 2)$;
    $mp\_flush\_cur\_exp(mp, new\_expr)$;
  } **while** $(cur\_cmd() \equiv mp\_comma)$;
}

1103.   ⟨Declare action procedures for use by $do\_statement$ 1048⟩ +≡
  static void $mp\_disp\_token(\mathbf{MP}\ mp)$;

1104.   **void** $mp\_disp\_token(\mathbf{MP}\ mp)$
{
  $mp\_print\_nl(mp, ">_\sqcup")$;
  ;
  if $(cur\_sym() \equiv \Lambda)$ {
    ⟨Show a numeric or string or capsule token 1105⟩;
  }
  else {
    $mp\_print\_text(cur\_sym())$;
    $mp\_print\_char(mp, xord('='))$;
    if $(eq\_type(cur\_sym()) \geq mp\_outer\_tag)$ $mp\_print(mp, "(outer)_\sqcup")$;
    $mp\_print\_cmd\_mod(mp, cur\_cmd(), cur\_mod())$;
    if $(cur\_cmd() \equiv mp\_defined\_macro)$ {
      $mp\_print\_ln(mp)$;
      $mp\_show\_macro(mp, cur\_mod\_node(), \Lambda, 100000)$;
    }   /* this avoids recursion between $show\_macro$ and $print\_cmd\_mod$ */
  }
}

1105. ⟨Show a numeric or string or capsule token 1105⟩ ≡
```
{
    if (cur_cmd( ) ≡ mp_numeric_token) {
        print_number(cur_mod_number( ));
    }
    else if (cur_cmd( ) ≡ mp_capsule_token) {
        mp_print_capsule(mp, cur_mod_node( ));
    }
    else {
        mp_print_char(mp, xord('"'));
        mp_print_str(mp, cur_mod_str( ));
        mp_print_char(mp, xord('"'));
        delete_str_ref(cur_mod_str( ));
    }
}
```
This code is used in section 1104.

1106.   The following cases of *print_cmd_mod* might arise in connection with *disp_token*, although they don't necessarily correspond to primitive tokens.

⟨Cases of *print_cmd_mod* for symbolic printing of primitives 233⟩ +≡
```
case mp_left_delimiter: case mp_right_delimiter:
    if (c ≡ mp_left_delimiter) mp_print(mp, "left");
    else mp_print(mp, "right");
#if 0
    mp_print(mp, "␣delimiter␣that␣matches␣");
    mp_print_text(m);
#else
    mp_print(mp, "␣delimiter");
#endif
    break;
case mp_tag_token:
    if (m ≡ 0)      /∗ todo: this was null ∗/
        mp_print(mp, "tag");
    else mp_print(mp, "variable");
    break;
case mp_defined_macro: mp_print(mp, "macro:");
    break;
case mp_secondary_primary_macro: case mp_tertiary_secondary_macro:
    case mp_expression_tertiary_macro: mp_print_cmd_mod(mp, mp_macro_def, c);
    mp_print(mp, "'d␣macro:");
    mp_print_ln(mp);
    mp_show_token_list(mp, mp_link(mp_link(cur_mod_node( ))), 0, 1000, 0);
    break;
case mp_repeat_loop: mp_print(mp, "[repeat␣the␣loop]");
    break;
case mp_internal_quantity: mp_print(mp, internal_name(m));
    break;
```

1107. ⟨Declare action procedures for use by *do_statement* 1048⟩ +≡
```
static void mp_do_show_token(MP mp);
```

1108.   **void** *mp_do_show_token*(**MP** *mp*)
{
  do {
    get_t_next(*mp*);
    mp_disp_token(*mp*);
    mp_get_x_next(*mp*);
  } **while** (*cur_cmd*( ) ≡ *mp_comma*);
}

1109.   ⟨Declare action procedures for use by *do_statement* 1048⟩ +≡
  static void *mp_do_show_stats*(**MP** *mp*);

1110.   **void** *mp_do_show_stats*(**MP** *mp*)
{
  mp_print_nl(*mp*, "Memory␣usage␣");
  ;
  mp_print_int(*mp*, (**integer**) *mp*→*var_used*);
  mp_print_ln(*mp*);
  mp_print_nl(*mp*, "String␣usage␣");
  mp_print_int(*mp*, (**int**) *mp*→*strs_in_use*);
  mp_print_char(*mp*, *xord*('&'));
  mp_print_int(*mp*, (**int**) *mp*→*pool_in_use*);
  mp_print_ln(*mp*);
  mp_get_x_next(*mp*);
}

1111.   Here's a recursive procedure that gives an abbreviated account of a variable, for use by *do_show_var*.

⟨Declare action procedures for use by *do_statement* 1048⟩ +≡
  static void *mp_disp_var*(**MP** *mp*, **mp_node** *p*);

1112.   **void** *mp_disp_var*(**MP** *mp*, **mp_node** *p*)
{
  mp_node *q*;   /∗ traverses attributes and subscripts ∗/
  int *n*;   /∗ amount of macro text to show ∗/
  if (*mp_type*(*p*) ≡ *mp_structured*) {
    ⟨Descend the structure 1113⟩;
  }
  else if (*mp_type*(*p*) ≥ *mp_unsuffixed_macro*) {
    ⟨Display a variable macro 1114⟩;
  }
  else if (*mp_type*(*p*) ≠ *mp_undefined*) {
    mp_print_nl(*mp*, "");
    mp_print_variable_name(*mp*, *p*);
    mp_print_char(*mp*, *xord*('='));
    mp_print_exp(*mp*, *p*, 0);
  }
}

1113.   ⟨Descend the structure 1113⟩ ≡
  {
    $q = attr\_head(p)$;
    do {
      $mp\_disp\_var(mp, q)$;
      $q = mp\_link(q)$;
    } **while** $(q \neq mp\text{→}end\_attr)$;
    $q = subscr\_head(p)$;
    while $(mp\_name\_type(q) \equiv mp\_subscr)$ {
      $mp\_disp\_var(mp, q)$;
      $q = mp\_link(q)$;
    }
  }

This code is used in section 1112.

1114.   ⟨Display a variable macro 1114⟩ ≡
  {
    $mp\_print\_nl(mp, \texttt{""})$;
    $mp\_print\_variable\_name(mp, p)$;
    if $(mp\_type(p) > mp\_unsuffixed\_macro)$ $mp\_print(mp, \texttt{"@#"})$;   /∗ *suffixed_macro* ∗/
    $mp\_print(mp, \texttt{"=macro:"})$;
    if $((\mathbf{int})\ mp\text{→}file\_offset \geq mp\text{→}max\_print\_line - 20)$ $n = 5$;
    else $n = mp\text{→}max\_print\_line - (\mathbf{int})\ mp\text{→}file\_offset - 15$;
    $mp\_show\_macro(mp, value\_node(p), \Lambda, n)$;
  }

This code is used in section 1112.

1115.   ⟨Declare action procedures for use by *do_statement* 1048⟩ +≡
  static void $mp\_do\_show\_var(\mathbf{MP}\ mp)$;

1116.   **void** $mp\_do\_show\_var(\mathbf{MP}\ mp)$
  {
    do {
      $get\_t\_next(mp)$;
      if $(cur\_sym() \neq \Lambda)$
        if $(cur\_sym\_mod() \equiv 0)$
          if $(cur\_cmd() \equiv mp\_tag\_token)$
            if $(cur\_mod() \neq 0)$ {
              $mp\_disp\_var(mp, cur\_mod\_node())$;
              goto DONE;
            }
      $mp\_disp\_token(mp)$;
    DONE: $mp\_get\_x\_next(mp)$;
    } **while** $(cur\_cmd() \equiv mp\_comma)$;
  }

1117.   ⟨Declare action procedures for use by *do_statement* 1048⟩ +≡
  static void $mp\_do\_show\_dependencies(\mathbf{MP}\ mp)$;

1118.   **void** $mp\_do\_show\_dependencies$(**MP** mp)
{
    mp_value_node p;   /∗ link that runs through all dependencies ∗/
    $p = ($**mp_value_node**$)\ mp\_link(mp\text{-}dep\_head)$;
    while $(p \neq mp\text{-}dep\_head)$ {
        if $(mp\_interesting(mp, ($**mp_node**$)\ p))$ {
            $mp\_print\_nl(mp, "")$;
            $mp\_print\_variable\_name(mp, ($**mp_node**$)\ p)$;
            if $(mp\_type(p) \equiv mp\_dependent)\ mp\_print\_char(mp, xord(\text{'='}))$;
            else $mp\_print(mp, "\text{␣=␣}")$;   /∗ extra spaces imply proto-dependency ∗/
            $mp\_print\_dependency(mp, ($**mp_value_node**$)\ dep\_list(p), mp\_type(p))$;
        }
        $p = ($**mp_value_node**$)\ dep\_list(p)$;
        while $(dep\_info(p) \neq \Lambda)\ p = ($**mp_value_node**$)\ mp\_link(p)$;
        $p = ($**mp_value_node**$)\ mp\_link(p)$;
    }
    $mp\_get\_x\_next(mp)$;
}

1119.   Finally we are ready for the procedure that governs all of the show commands.

⟨ Declare action procedures for use by $do\_statement$ 1048 ⟩ +≡
    static void $mp\_do\_show\_whatever$(**MP** mp);

1120.   **void** $mp\_do\_show\_whatever(\mathbf{MP}\ mp)$
  {
    if $(mp\rightarrow interaction \equiv mp\_error\_stop\_mode)$ $wake\_up\_terminal(\,)$;
    switch $(cur\_mod(\,))$ {
    case $show\_token\_code$: $mp\_do\_show\_token(mp)$;
      break;
    case $show\_stats\_code$: $mp\_do\_show\_stats(mp)$;
      break;
    case $show\_code$: $mp\_do\_show(mp)$;
      break;
    case $show\_var\_code$: $mp\_do\_show\_var(mp)$;
      break;
    case $show\_dependencies\_code$: $mp\_do\_show\_dependencies(mp)$;
      break;
    }   /* there are no other cases */
    if $(number\_positive(internal\_value(mp\_showstopping)))$ {
      const char $*hlp[\,] = \{$"This␣isn't␣an␣error␣message;␣I'm␣just␣showing␣something.", $\Lambda\}$;

      if $(mp\rightarrow interaction < mp\_error\_stop\_mode)$ {
        $hlp[0] = \Lambda$;
        $decr(mp\rightarrow error\_count)$;
      }
      if $(cur\_cmd(\,) \equiv mp\_semicolon)$ {
        $mp\_error(mp,$ "OK"$, hlp, true)$;
      }
      else {
        $mp\_back\_error(mp,$ "OK"$, hlp, true)$;
        $mp\_get\_x\_next(mp)$;
      }
      ;
    }
  }

1121.   The '**addto**' command needs the following additional primitives:

#define $double\_path\_code$ 0   /* command modifier for '**doublepath**' */
#define $contour\_code$ 1   /* command modifier for '**contour**' */
#define $also\_code$ 2   /* command modifier for '**also**' */

1122.   Pre and postscripts need two new identifiers:

#define *with_mp_pre_script*   11
#define *with_mp_post_script*   13

⟨ Put each of METAPOST's primitives into the hash table 200 ⟩ +≡
  mp_primitive(*mp*, "doublepath", *mp_thing_to_add*, *double_path_code*);
  ;
  mp_primitive(*mp*, "contour", *mp_thing_to_add*, *contour_code*);
  ;
  mp_primitive(*mp*, "also", *mp_thing_to_add*, *also_code*);
  ;
  mp_primitive(*mp*, "withpen", *mp_with_option*, *mp_pen_type*);
  ;
  mp_primitive(*mp*, "dashed", *mp_with_option*, *mp_picture_type*);
  ;
  mp_primitive(*mp*, "withprescript", *mp_with_option*, *with_mp_pre_script*);
  ;
  mp_primitive(*mp*, "withpostscript", *mp_with_option*, *with_mp_post_script*);
  ;
  mp_primitive(*mp*, "withoutcolor", *mp_with_option*, *mp_no_model*);
  ;
  mp_primitive(*mp*, "withgreyscale", *mp_with_option*, *mp_grey_model*);
  ;
  mp_primitive(*mp*, "withcolor", *mp_with_option*, *mp_uninitialized_model*);
     /∗ **withrgbcolor** is an alias for **withcolor** ∗/
  mp_primitive(*mp*, "withrgbcolor", *mp_with_option*, *mp_rgb_model*);
  ;
  mp_primitive(*mp*, "withcmykcolor", *mp_with_option*, *mp_cmyk_model*);

1123.   ⟨ Cases of *print_cmd_mod* for symbolic printing of primitives 233 ⟩ +≡
case *mp_thing_to_add*:
  if (*m* ≡ *contour_code*) *mp_print*(*mp*, "contour");
  else if (*m* ≡ *double_path_code*) *mp_print*(*mp*, "doublepath");
  else *mp_print*(*mp*, "also");
  break;
case *mp_with_option*:
  if (*m* ≡ *mp_pen_type*) *mp_print*(*mp*, "withpen");
  else if (*m* ≡ *with_mp_pre_script*) *mp_print*(*mp*, "withprescript");
  else if (*m* ≡ *with_mp_post_script*) *mp_print*(*mp*, "withpostscript");
  else if (*m* ≡ *mp_no_model*) *mp_print*(*mp*, "withoutcolor");
  else if (*m* ≡ *mp_rgb_model*) *mp_print*(*mp*, "withrgbcolor");
  else if (*m* ≡ *mp_uninitialized_model*) *mp_print*(*mp*, "withcolor");
  else if (*m* ≡ *mp_cmyk_model*) *mp_print*(*mp*, "withcmykcolor");
  else if (*m* ≡ *mp_grey_model*) *mp_print*(*mp*, "withgreyscale");
  else *mp_print*(*mp*, "dashed");
  break;

1124.   The *scan_with_list* procedure parses a ⟨with list⟩ and updates the list of graphical objects starting
at *p*. Each ⟨with clause⟩ updates all graphical objects whose *type* is compatible. Other objects are ignored.

⟨ Declare action procedures for use by *do_statement* 1048 ⟩ +≡
  static void *mp_scan_with_list*(**MP** *mp*, **mp_node** *p*);

1125.   Forcing the color to be between 0 and *unity* here guarantees that no picture will ever contain a color outside the legal range for PostScript graphics.

#define *make_cp_a_colored_object*() **do**
      {
        cp = *p*;
        while (*cp* ≠ Λ) {
          if (*has_color*(*cp*)) **break**;
          cp = *mp_link*(*cp*);
        }
      }
      while (0)
#define *clear_color*(*A*) **do**
      {
        set_number_to_zero(((**mp_stroked_node**)(*A*))→*cyan*);
        set_number_to_zero(((**mp_stroked_node**)(*A*))→*magenta*);
        set_number_to_zero(((**mp_stroked_node**)(*A*))→*yellow*);
        set_number_to_zero(((**mp_stroked_node**)(*A*))→*black*);
        mp_color_model((*A*)) = *mp_uninitialized_model*;
      }
      while (0)
#define *set_color_val*(*A*, *B*) **do**
      {
        number_clone(*A*, (*B*));
        if (*number_negative*(*A*)) *set_number_to_zero*(*A*);
        if (*number_greater*(*A*, *unity_t*)) *set_number_to_unity*(*A*);
      }
      while (0)
  static int *is_invalid_with_list*(**MP** *mp*, *mp_variable_type* *t*)
  {
    return ((*t* ≡ *with_mp_pre_script*) ∧ (*mp*→*cur_exp*.*type* ≠ *mp_string_type*)) ∨ ((*t* ≡
        with_mp_post_script) ∧ (*mp*→*cur_exp*.*type* ≠ *mp_string_type*)) ∨ ((*t* ≡
        (*mp_variable_type*)*mp_uninitialized_model*) ∧ ((*mp*→*cur_exp*.*type* ≠ *mp_cmykcolor_type*) ∧
        (*mp*→*cur_exp*.*type* ≠ *mp_color_type*) ∧ (*mp*→*cur_exp*.*type* ≠ *mp_known*) ∧ (*mp*→*cur_exp*.*type* ≠
        mp_boolean_type))) ∨ ((*t* ≡ (*mp_variable_type*)*mp_cmyk_model*) ∧ (*mp*→*cur_exp*.*type* ≠
        mp_cmykcolor_type)) ∨ ((*t* ≡ (*mp_variable_type*)*mp_rgb_model*) ∧ (*mp*→*cur_exp*.*type* ≠
        mp_color_type)) ∨ ((*t* ≡ (*mp_variable_type*)*mp_grey_model*) ∧ (*mp*→*cur_exp*.*type* ≠
        mp_known)) ∨ ((*t* ≡ (*mp_variable_type*)*mp_pen_type*) ∧ (*mp*→*cur_exp*.*type* ≠ *t*)) ∨ ((*t* ≡
        (*mp_variable_type*)*mp_picture_type*) ∧ (*mp*→*cur_exp*.*type* ≠ *t*));
  }
  static void *complain_invalid_with_list*(**MP** *mp*, *mp_variable_type* *t*)
  {   /∗ Complain about improper type ∗/
  mp_value *new_expr*;
  const char ∗*hlp*[] = {"Next␣time␣say␣'withpen␣<known␣pen␣expression>';",
    "I'll␣ignore␣the␣bad␣'with'␣clause␣and␣look␣for␣another.", Λ};
  memset(&*new_expr*, 0, **sizeof**(**mp_value**));
  new_number(*new_expr*.*data*.*n*);
  mp_disp_err(*mp*, Λ);
  if (*t* ≡ *with_mp_pre_script*)
    hlp[0] = "Next␣time␣say␣'withprescript␣<known␣string␣expression>';";
  else if (*t* ≡ *with_mp_post_script*)
    hlp[0] = "Next␣time␣say␣'withpostscript␣<known␣string␣expression>';";

    else if $(t \equiv mp\_picture\_type)$ $hlp[0] = $ "Next␣time␣say␣'dashed␣<known␣picture␣expression>';";
    else if $(t \equiv (mp\_variable\_type)mp\_uninitialized\_model)$
        $hlp[0] = $ "Next␣time␣say␣'withcolor␣<known␣color␣expression>';";
    else if $(t \equiv (mp\_variable\_type)mp\_rgb\_model)$
        $hlp[0] = $ "Next␣time␣say␣'withrgbcolor␣<known␣color␣expression>';";
    else if $(t \equiv (mp\_variable\_type)mp\_cmyk\_model)$
        $hlp[0] = $ "Next␣time␣say␣'withcmykcolor␣<known␣cmykcolor␣expression>';";
    else if $(t \equiv (mp\_variable\_type)mp\_grey\_model)$
        $hlp[0] = $ "Next␣time␣say␣'withgreyscale␣<known␣numeric␣expression>';";
    ;
    $mp\_back\_error(mp,$ "Improper␣type", $hlp, true);$
    $mp\_get\_x\_next(mp);$
    $mp\_flush\_cur\_exp(mp, new\_expr);$
}
void $mp\_scan\_with\_list(\textbf{MP}\ mp, \textbf{mp\_node}\ p)$
{
    $mp\_variable\_type t;$ /∗ cur_mod of the $with\_option$ (should match cur_type) ∗/

    mp_node $q;$ /∗ for list manipulation ∗/
    mp_node $cp, pp, dp, ap, bp;$ /∗ objects being updated; **void** initially; Λ to suppress update ∗/

    $cp = $ MP_VOID;
    $pp = $ MP_VOID;
    $dp = $ MP_VOID;
    $ap = $ MP_VOID;
    $bp = $ MP_VOID;
    while $(cur\_cmd() \equiv mp\_with\_option)$ {
        /∗ todo this is not very nice: the color models have their own enumeration ∗/
        $t = (mp\_variable\_type)cur\_mod();$
        $mp\_get\_x\_next(mp);$
        if $(t \neq (mp\_variable\_type)mp\_no\_model)$ $mp\_scan\_expression(mp);$
        if $(is\_invalid\_with\_list(mp, t))$ {
            $complain\_invalid\_with\_list(mp, t);$
            continue;
        }
        if $(t \equiv (mp\_variable\_type)mp\_uninitialized\_model)$ {
            mp_value $new\_expr;$

            $memset(\&new\_expr, 0, \textbf{sizeof}(\textbf{mp\_value}));$
            $new\_number(new\_expr.data.n);$
            if $(cp \equiv $ MP_VOID$)$ $make\_cp\_a\_colored\_object();$
            if $(cp \neq \Lambda)$ {   /∗ Transfer a color from the current expression to object cp ∗/
                if $(mp\text{-}cur\_exp.type \equiv mp\_color\_type)$ {
                    /∗ Transfer a rgbcolor from the current expression to object cp ∗/
                    mp_stroked_node $cp0 = (\textbf{mp\_stroked\_node})\ cp;$

                    $q = value\_node(cur\_exp\_node());$
                    $clear\_color(cp0);$
                    mp_color_model$(cp) = mp\_rgb\_model;$
                    $set\_color\_val(cp0\text{-}red, value\_number(red\_part(q)));$
                    $set\_color\_val(cp0\text{-}green, value\_number(green\_part(q)));$
                    $set\_color\_val(cp0\text{-}blue, value\_number(blue\_part(q)));$
                }
                else if $(mp\text{-}cur\_exp.type \equiv mp\_cmykcolor\_type)$ {
                    /∗ Transfer a cmykcolor from the current expression to object cp ∗/

$\quad$ **mp_stroked_node** $cp0 = ($**mp_stroked_node**$) \; cp$;

$\qquad q = value\_node(cur\_exp\_node(\,))$;
$\qquad set\_color\_val(cp0\rightarrow cyan, value\_number(cyan\_part(q)))$;
$\qquad set\_color\_val(cp0\rightarrow magenta, value\_number(magenta\_part(q)))$;
$\qquad set\_color\_val(cp0\rightarrow yellow, value\_number(yellow\_part(q)))$;
$\qquad set\_color\_val(cp0\rightarrow black, value\_number(black\_part(q)))$;
$\qquad$ **mp_color_model**$(cp) = mp\_cmyk\_model$;
$\quad$ }
$\quad$ **else if** $(mp\rightarrow cur\_exp.type \equiv mp\_known)$ {
$\qquad$ /∗ Transfer a greyscale from the current expression to object cp ∗/
$\qquad$ **mp_number** qq;
$\qquad$ **mp_stroked_node** $cp0 = ($**mp_stroked_node**$) \; cp$;

$\qquad new\_number(qq)$;
$\qquad number\_clone(qq, cur\_exp\_value\_number(\,))$;
$\qquad clear\_color(cp)$;
$\qquad$ **mp_color_model**$(cp) = mp\_grey\_model$;
$\qquad set\_color\_val(cp0\rightarrow grey, qq)$;
$\qquad free\_number(qq)$;
$\quad$ }
$\quad$ **else if** $(cur\_exp\_value\_boolean(\,) \equiv mp\_false\_code)$ {
$\qquad$ /∗ Transfer a noncolor from the current expression to object cp ∗/
$\qquad clear\_color(cp)$;
$\qquad$ **mp_color_model**$(cp) = mp\_no\_model$;
$\quad$ }
$\quad$ **else if** $(cur\_exp\_value\_boolean(\,) \equiv mp\_true\_code)$ {
$\qquad$ /∗ Transfer no color from the current expression to object cp ∗/
$\qquad clear\_color(cp)$;
$\qquad$ **mp_color_model**$(cp) = mp\_uninitialized\_model$;
$\quad$ }
$\quad$ }
$\quad mp\_flush\_cur\_exp(mp, new\_expr)$;
}
else if $(t \equiv (mp\_variable\_type)mp\_rgb\_model)$ {
$\quad$ **mp_value** new_expr;

$\quad memset(\&new_expr, 0, $**sizeof**$($**mp_value**$))$;
$\quad new\_number(new\_expr.data.n)$;
$\quad$ **if** $(cp \equiv $ MP_VOID$) \; make\_cp\_a\_colored\_object(\,)$;
$\quad$ **if** $(cp \neq \Lambda)$ {$\qquad$ /∗ Transfer a rgbcolor from the current expression to object cp ∗/
$\qquad$ **mp_stroked_node** $cp0 = ($**mp_stroked_node**$) \; cp$;

$\qquad q = value\_node(cur\_exp\_node(\,))$;
$\qquad clear\_color(cp0)$;
$\qquad$ **mp_color_model**$(cp) = mp\_rgb\_model$;
$\qquad set\_color\_val(cp0\rightarrow red, value\_number(red\_part(q)))$;
$\qquad set\_color\_val(cp0\rightarrow green, value\_number(green\_part(q)))$;
$\qquad set\_color\_val(cp0\rightarrow blue, value\_number(blue\_part(q)))$;
$\quad$ }
$\quad mp\_flush\_cur\_exp(mp, new\_expr)$;
}
else if $(t \equiv (mp\_variable\_type)mp\_cmyk\_model)$ {
$\quad$ **mp_value** new_expr;

$\quad memset(\&new_expr, 0, $**sizeof**$($**mp_value**$))$;

$new\_number(new\_expr.data.n);$

if $(cp \equiv \texttt{MP\_VOID})$ $make\_cp\_a\_colored\_object()$;

if $(cp \neq \Lambda)$ {   /* Transfer a cmykcolor from the current expression to object cp */

  mp_stroked_node $cp0 = (\textbf{mp\_stroked\_node})$ cp;

  $q = value\_node(cur\_exp\_node());$
  $set\_color\_val(cp0\rightarrow cyan, value\_number(cyan\_part(q)));$
  $set\_color\_val(cp0\rightarrow magenta, value\_number(magenta\_part(q)));$
  $set\_color\_val(cp0\rightarrow yellow, value\_number(yellow\_part(q)));$
  $set\_color\_val(cp0\rightarrow black, value\_number(black\_part(q)));$
  mp_color_model$(cp) = mp\_cmyk\_model$;

}

$mp\_flush\_cur\_exp(mp, new\_expr);$

}

else if $(t \equiv (mp\_variable\_type)mp\_grey\_model)$ {

  mp_value new_expr;

  $memset(\&new\_expr, 0, \textbf{sizeof}(\textbf{mp\_value}));$
  $new\_number(new\_expr.data.n);$

  if $(cp \equiv \texttt{MP\_VOID})$ $make\_cp\_a\_colored\_object()$;

  if $(cp \neq \Lambda)$ {   /* Transfer a greyscale from the current expression to object cp */

    mp_number qq;

    mp_stroked_node $cp0 = (\textbf{mp\_stroked\_node})$ cp;

    $new\_number(qq);$
    $number\_clone(qq, cur\_exp\_value\_number());$
    $clear\_color(cp);$
    mp_color_model$(cp) = mp\_grey\_model$;
    $set\_color\_val(cp0\rightarrow grey, qq);$
    $free\_number(qq);$

  }

  $mp\_flush\_cur\_exp(mp, new\_expr);$

}

else if $(t \equiv (mp\_variable\_type)mp\_no\_model)$ {

  if $(cp \equiv \texttt{MP\_VOID})$ $make\_cp\_a\_colored\_object()$;

  if $(cp \neq \Lambda)$ {   /* Transfer a noncolor from the current expression to object cp */

    $clear\_color(cp);$
    mp_color_model$(cp) = mp\_no\_model$;

  }

}

else if $(t \equiv mp\_pen\_type)$ {

  if $(pp \equiv \texttt{MP\_VOID})$ {   /* Make pp an object in list p that needs a pen */

    $pp = p$;

    while $(pp \neq \Lambda)$ {

      if $(has\_pen(pp))$ **break**;

      $pp = mp\_link(pp);$

    }

  }

  if $(pp \neq \Lambda)$ {

    switch $(mp\_type(pp))$ {

    case $mp\_fill\_node\_type$:

      if $(mp\_pen\_p((\textbf{mp\_fill\_node})$ $pp) \neq \Lambda)$

        $mp\_toss\_knot\_list(mp, mp\_pen\_p((\textbf{mp\_fill\_node})$ $pp));$

      $mp\_pen\_p((\textbf{mp\_fill\_node})$ $pp) = cur\_exp\_knot();$

```
              break;
          case mp_stroked_node_type:
            if (mp_pen_p((mp_stroked_node) pp) ≠ Λ)
                mp_toss_knot_list(mp, mp_pen_p((mp_stroked_node) pp));
            mp_pen_p((mp_stroked_node) pp) = cur_exp_knot();
              break;
          default: assert(0);
              break;
          }
          mp→cur_exp.type = mp_vacuous;
        }
      }
      else if (t ≡ with_mp_pre_script) {
        if (cur_exp_str()→len) {
          if (ap ≡ MP_VOID) ap = p;
          while ((ap ≠ Λ) ∧ (¬has_color(ap))) ap = mp_link(ap);
          if (ap ≠ Λ) {
            if (mp_pre_script(ap) ≠ Λ) {      /* build a new,combined string */
                unsigned old_setting;       /* saved selector setting */
                mp_string s;        /* for string cleanup after combining */

                s = mp_pre_script(ap);
                old_setting = mp→selector;
                mp→selector = new_string;
                str_room(mp_pre_script(ap)→len + cur_exp_str()→len + 2);
                mp_print_str(mp, cur_exp_str());
                append_char(13);        /* a forced PostScript newline */
                mp_print_str(mp, mp_pre_script(ap));
                mp_pre_script(ap) = mp_make_string(mp);
                delete_str_ref(s);
                mp→selector = old_setting;
            }
            else {
                mp_pre_script(ap) = cur_exp_str();
            }
            add_str_ref(mp_pre_script(ap));
            mp→cur_exp.type = mp_vacuous;
          }
        }
      }
      else if (t ≡ with_mp_post_script) {
        if (cur_exp_str()→len) {
          mp_node k = Λ;        /* for finding the near-last item in a list */
          if (bp ≡ MP_VOID) k = p;
          bp = k;
          while (k ∧ mp_link(k) ≠ Λ) {      /* clang: dereference null pointer 'k' */
            k = mp_link(k);
            if (has_color(k)) bp = k;
          }
          if (bp ≠ Λ) {
            if (mp_post_script(bp) ≠ Λ) {
                unsigned old_setting;       /* saved selector setting */
                mp_string s;       /* for string cleanup after combining */
```

```
            s = mp_post_script(bp);
            old_setting = mp→selector;
            mp→selector = new_string;
            str_room(mp_post_script(bp)→len + cur_exp_str()→len + 2);
            mp_print_str(mp, mp_post_script(bp));
            append_char(13);      /* a forced PostScript newline */
            mp_print_str(mp, cur_exp_str());
            mp_post_script(bp) = mp_make_string(mp);
            delete_str_ref(s);
            mp→selector = old_setting;
          }
          else {
            mp_post_script(bp) = cur_exp_str();
          }
          add_str_ref(mp_post_script(bp));
          mp→cur_exp.type = mp_vacuous;
        }
      }
    }
    else {
      if (dp ≡ MP_VOID) {      /* Make dp a stroked node in list p */
        dp = p;
        while (dp ≠ Λ) {
          if (mp_type(dp) ≡ mp_stroked_node_type) break;
          dp = mp_link(dp);
        }
      }
      if (dp ≠ Λ) {
        if (mp_dash_p(dp) ≠ Λ) delete_edge_ref(mp_dash_p(dp));
        mp_dash_p(dp) = (mp_node) mp_make_dashes(mp, (mp_edge_header_node) cur_exp_node());
        set_number_to_unity(((mp_stroked_node) dp)→dash_scale);
        mp→cur_exp.type = mp_vacuous;
      }
    }
  }      /* Copy the information from objects cp, pp, and dp into the rest of the list */
  if (cp > MP_VOID) {      /* Copy cp's color into the colored objects linked to cp */
    q = mp_link(cp);
    while (q ≠ Λ) {
      if (has_color(q)) {
        mp_stroked_node q0 = (mp_stroked_node) q;
        mp_stroked_node cp0 = (mp_stroked_node) cp;

        number_clone(q0→red, cp0→red);
        number_clone(q0→green, cp0→green);
        number_clone(q0→blue, cp0→blue);
        number_clone(q0→black, cp0→black);
        mp_color_model(q) = mp_color_model(cp);
      }
      q = mp_link(q);
    }
  }
  if (pp > MP_VOID) {      /* Copy mp_pen_p(pp) into stroked and filled nodes linked to pp */
    q = mp_link(pp);
```

```
    while (q ≠ Λ) {
      if (has_pen(q)) {
        switch (mp_type(q)) {
        case mp_fill_node_type:
          if (mp_pen_p((mp_fill_node) q) ≠ Λ) mp_toss_knot_list(mp, mp_pen_p((mp_fill_node) q));
          mp_pen_p((mp_fill_node) q) = copy_pen(mp_pen_p((mp_fill_node) pp));
          break;
        case mp_stroked_node_type:
          if (mp_pen_p((mp_stroked_node) q) ≠ Λ)
            mp_toss_knot_list(mp, mp_pen_p((mp_stroked_node) q));
          mp_pen_p((mp_stroked_node) q) = copy_pen(mp_pen_p((mp_stroked_node) pp));
          break;
        default: assert(0);
          break;
        }
      }
      q = mp_link(q);
    }
  }
  if (dp > MP_VOID) {      /* Make stroked nodes linked to dp refer to mp_dash_p(dp) */
    q = mp_link(dp);
    while (q ≠ Λ) {
      if (mp_type(q) ≡ mp_stroked_node_type) {
        if (mp_dash_p(q) ≠ Λ) delete_edge_ref(mp_dash_p(q));
        mp_dash_p(q) = mp_dash_p(dp);
        set_number_to_unity(((mp_stroked_node) q)→dash_scale);
        if (mp_dash_p(q) ≠ Λ) add_edge_ref(mp_dash_p(q));
      }
      q = mp_link(q);
    }
  }
}
```

1126. One of the things we need to do when we've parsed an **addto** or similar command is find the header of a supposed **picture** variable, given a token list for that variable. Since the edge structure is about to be updated, we use *private_edges* to make sure that this is possible.

⟨ Declare action procedures for use by *do_statement* 1048 ⟩ +≡
  static mp_edge_header_node *mp_find_edges_var*(**MP** *mp*, **mp_node** *t*);

1127.   **mp_edge_header_node** *mp_find_edges_var*(**MP** *mp*, **mp_node** *t*)
  {
      mp_node *p*;
      mp_edge_header_node *cur_edges*;   /∗ the return value ∗/
      p = *mp_find_variable*(*mp*, *t*);
      cur_edges = Λ;
      if (*p* ≡ Λ) {
          const char ∗*hlp*[ ] = {"It␣seems␣you␣did␣a␣nasty␣thing−−−probably␣by␣accident,",
              "but␣nevertheless␣you␣nearly␣hornswoggled␣me...",
              "While␣I␣was␣evaluating␣the␣right−hand␣side␣of␣this",
              "command,␣something␣happened,␣and␣the␣left−hand␣side",
              "is␣no␣longer␣a␣variable!␣So␣I␣won't␣change␣anything.", Λ};
          char ∗*msg* = *mp_obliterated*(*mp*, *t*);
          mp_back_error(*mp*, *msg*, *hlp*, *true*);
          free(*msg*);
          mp_get_x_next(*mp*);
      }
      else if (*mp_type*(*p*) ≠ *mp_picture_type*) {
          char *msg*[256];
          mp_string *sname*;
          int *old_setting* = *mp*→*selector*;
          const char ∗*hlp*[ ] = {"I␣was␣looking␣for␣a␣\"known\"␣picture␣variable.",
              "So␣I'll␣not␣change␣anything␣just␣now.", Λ};
          mp→*selector* = *new_string*;
          mp_show_token_list(*mp*, *t*, Λ, 1000, 0);
          sname = *mp_make_string*(*mp*);
          mp→*selector* = *old_setting*;
          mp_snprintf(*msg*, 256, "Variable␣%s␣is␣the␣wrong␣type(%s)", *mp_str*(*mp*, *sname*),
              mp_type_string(*mp_type*(*p*)));
          ;
          delete_str_ref(*sname*);
          mp_back_error(*mp*, *msg*, *hlp*, *true*);
          mp_get_x_next(*mp*);
      }
      else {
          set_value_node(*p*, (**mp_node**) *mp_private_edges*(*mp*, (**mp_edge_header_node**) *value_node*(*p*)));
          cur_edges = (**mp_edge_header_node**) *value_node*(*p*);
      }
      mp_flush_node_list(*mp*, *t*);
      return *cur_edges*;
  }

1128.   ⟨ Put each of METAPOST's primitives into the hash table 200 ⟩ +≡
  mp_primitive(*mp*, "clip", *mp_bounds_command*, *mp_start_clip_node_type*);
  ;
  mp_primitive(*mp*, "setbounds", *mp_bounds_command*, *mp_start_bounds_node_type*);

1129. ⟨Cases of *print_cmd_mod* for symbolic printing of primitives 233⟩ +≡
case *mp_bounds_command*:
  if (*m* ≡ *mp_start_clip_node_type*) *mp_print*(*mp*, "clip");
  else *mp_print*(*mp*, "setbounds");
  break;

1130. The following function parses the beginning of an **addto** or **clip** command: it expects a variable
name followed by a token with *cur_cmd = sep* and then an expression. The function returns the token list
for the variable and stores the command modifier for the separator token in the global variable *last_add_type*.
We must be careful because this variable might get overwritten any time we call *get_x_next*.

⟨Global variables 14⟩ +≡
  quarterword *last_add_type*;   /* command modifier that identifies the last **addto** command */

1131. ⟨Declare action procedures for use by *do_statement* 1048⟩ +≡
  static mp_node *mp_start_draw_cmd*(**MP** *mp*, **quarterword** *sep*);

1132.   **mp_node** *mp_start_draw_cmd*(**MP** *mp*, **quarterword** *sep*)
  {
    mp_node *lhv*;   /* variable to add to left */
    quarterword *add_type* = 0;   /* value to be returned in *last_add_type* */
    lhv = Λ;
    mp_get_x_next(*mp*);
    mp→var_flag = *sep*;
    mp_scan_primary(*mp*);
    if (*mp→cur_exp.type* ≠ *mp_token_list*) {
      /* Abandon edges command because there's no variable */
      mp_value *new_expr*;
      const char *hlp[] = {"At␣this␣point␣I␣needed␣to␣see␣the␣name␣of␣a␣picture␣variable.",
        "(Or␣perhaps␣you␣have␣indeed␣presented␣me␣with␣one;␣I␣might",
        "have␣missed␣it,␣if␣it␣wasn't␣followed␣by␣the␣proper␣token.)",
        "So␣I'll␣not␣change␣anything␣just␣now.", Λ};
      memset(&*new_expr*, 0, **sizeof**(**mp_value**));
      new_number(*new_expr.data.n*);
      mp_disp_err(*mp*, Λ);
      set_number_to_zero(*new_expr.data.n*);
      mp_back_error(*mp*, "Not␣a␣suitable␣variable", *hlp*, *true*);
      mp_get_x_next(*mp*);
      mp_flush_cur_exp(*mp*, *new_expr*);
    }
    else {
      lhv = *cur_exp_node*();
      add_type = (**quarterword**) *cur_mod*();
      mp→cur_exp.type = *mp_vacuous*;
      mp_get_x_next(*mp*);
      mp_scan_expression(*mp*);
    }
    mp→last_add_type = *add_type*;
    return *lhv*;
  }

1133.   Here is an example of how to use *start_draw_cmd*.

⟨ Declare action procedures for use by *do_statement* 1048 ⟩ +≡
  static void *mp_do_bounds*(**MP** *mp*);

1134.   **void** mp_do_bounds(**MP** mp)
{
    mp_node lhv;   /∗ variable on left, the corresponding edge structure ∗/
    mp_edge_header_node lhe;
    mp_node p;   /∗ for list manipulation ∗/
    integer m;   /∗ initial value of cur_mod ∗/
    $m = cur\_mod(\,)$;
    $lhv = mp\_start\_draw\_cmd(mp, mp\_to\_token)$;
    if $(lhv \neq \Lambda)$ {
        mp_value new_expr;

        $memset(\&new\_expr, 0, \mathbf{sizeof}(\mathbf{mp\_value}))$;
        $lhe = mp\_find\_edges\_var(mp, lhv)$;
        if $(lhe \equiv \Lambda)$ {
            $new\_number(new\_expr.data.n)$;
            $set\_number\_to\_zero(new\_expr.data.n)$;
            $mp\_flush\_cur\_exp(mp, new\_expr)$;
        }
        else if $(mp\text{→}cur\_exp.type \neq mp\_path\_type)$ {
            const char $∗hlp[\,] = \{$"This␣expression␣should␣have␣specified␣a␣known␣path.",
                "So␣I'll␣not␣change␣anything␣just␣now.", $\Lambda\}$;

            $mp\_disp\_err(mp, \Lambda)$;
            $new\_number(new\_expr.data.n)$;
            $set\_number\_to\_zero(new\_expr.data.n)$;
            $mp\_back\_error(mp,$ "Improper␣'clip'", $hlp, true)$;
            $mp\_get\_x\_next(mp)$;
            $mp\_flush\_cur\_exp(mp, new\_expr)$;
        }
        else if $(mp\_left\_type(cur\_exp\_knot(\,)) \equiv mp\_endpoint)$ {   /∗ Complain about a non-cycle ∗/
            const char $∗hlp[\,] = \{$"That␣contour␣should␣have␣ended␣with␣'..cycle'␣or␣'&cycle'.",
                "So␣I'll␣not␣change␣anything␣just␣now.", $\Lambda\}$;

            $mp\_back\_error(mp,$ "Not␣a␣cycle", $hlp, true)$;
            $mp\_get\_x\_next(mp)$;
        }
        else {   /∗ Make cur_exp into a **setbounds** or clipping path and add it to lhe ∗/
            $p = mp\_new\_bounds\_node(mp, cur\_exp\_knot(\,), (\mathbf{quarterword})\,m)$;
            $mp\_link(p) = mp\_link(edge\_list(lhe))$;
            $mp\_link(edge\_list(lhe)) = p$;
            if $(obj\_tail(lhe) \equiv edge\_list(lhe))$ $obj\_tail(lhe) = p$;
            if $(m \equiv mp\_start\_clip\_node\_type)$ {
                $p = mp\_new\_bounds\_node(mp, \Lambda, mp\_stop\_clip\_node\_type)$;
            }
            else if $(m \equiv mp\_start\_bounds\_node\_type)$ {
                $p = mp\_new\_bounds\_node(mp, \Lambda, mp\_stop\_bounds\_node\_type)$;
            }
            $mp\_link(obj\_tail(lhe)) = p$;
            $obj\_tail(lhe) = p$;
            $mp\_init\_bbox(mp, lhe)$;
         }
        }
    }
}

1135.   The *do_add_to* procedure is a little like *do_clip* but there are a lot more cases to deal with.

⟨ Declare action procedures for use by *do_statement* 1048 ⟩ +≡
  static void *mp_do_add_to*(**MP** *mp*);

1136.   **void** $mp\_do\_add\_to(\mathbf{MP}\ mp)$
$\{$
    $\mathbf{mp\_node}\ lhv;$
    $\mathbf{mp\_edge\_header\_node}\ lhe;$ /∗ variable on left, the corresponding edge structure ∗/
    $\mathbf{mp\_node}\ p;$ /∗ the graphical object or list for $scan\_with\_list$ to update ∗/
    $\mathbf{mp\_edge\_header\_node}\ e;$ /∗ an edge structure to be merged ∗/
    $\mathbf{quarterword}\ add\_type;$ /∗ $also\_code$, $contour\_code$, or $double\_path\_code$ ∗/

    $lhv = mp\_start\_draw\_cmd(mp, mp\_thing\_to\_add);$
    $add\_type = mp\text{-}last\_add\_type;$
    if $(lhv \neq \Lambda)\ \{$
        if $(add\_type \equiv also\_code)\ \{$ /∗ Make sure the current expression is a suitable picture and set e
                and p appropriately ∗/   /∗ Setting p: $= \Lambda$ causes the ⟨with list⟩ to be ignored; setting e:
                $= \Lambda$ prevents anything from being added to lhe. ∗/
            $p = \Lambda;$
            $e = \Lambda;$
            if $(mp\text{-}cur\_exp.type \neq mp\_picture\_type)\ \{$
                $\mathbf{mp\_value}\ new\_expr;$
                $\mathbf{const\ char}\ \ast hlp[\,] = \{\texttt{"This\textvisiblespace expression\textvisiblespace should\textvisiblespace have\textvisiblespace specified\textvisiblespace a\textvisiblespace known\textvisiblespace picture."},$
                    $\texttt{"So\textvisiblespace I'll\textvisiblespace not\textvisiblespace change\textvisiblespace anything\textvisiblespace just\textvisiblespace now."}, \Lambda\};$
                $memset(\&new\_expr, 0, \mathbf{sizeof}(\mathbf{mp\_value}));$
                $new\_number(new\_expr.data.n);$
                $mp\_disp\_err(mp, \Lambda);$
                $set\_number\_to\_zero(new\_expr.data.n);$
                $mp\_back\_error(mp, \texttt{"Improper\textvisiblespace `addto'"}, hlp, true);$
                $mp\_get\_x\_next(mp);$
                $mp\_flush\_cur\_exp(mp, new\_expr);$
            $\}$
            else $\{$
                $e = mp\_private\_edges(mp, (\mathbf{mp\_edge\_header\_node})\ cur\_exp\_node(\,));$
                $mp\text{-}cur\_exp.type = mp\_vacuous;$
                $p = mp\_link(edge\_list(e));$
            $\}$
        $\}$
        else $\{$ /∗ Create a graphical object p based on add_type and the current expression ∗/
            /∗ In this case $add\_type <> also\_code$ so setting p: $= \Lambda$ suppresses future attempts to add to
                the edge structure. ∗/
            $e = \Lambda;$
            $p = \Lambda;$
            if $(mp\text{-}cur\_exp.type \equiv mp\_pair\_type)\ mp\_pair\_to\_path(mp);$
            if $(mp\text{-}cur\_exp.type \neq mp\_path\_type)\ \{$
                $\mathbf{mp\_value}\ new\_expr;$
                $\mathbf{const\ char}\ \ast hlp[\,] = \{\texttt{"This\textvisiblespace expression\textvisiblespace should\textvisiblespace have\textvisiblespace specified\textvisiblespace a\textvisiblespace known\textvisiblespace path."},$
                    $\texttt{"So\textvisiblespace I'll\textvisiblespace not\textvisiblespace change\textvisiblespace anything\textvisiblespace just\textvisiblespace now."}, \Lambda\};$
                $memset(\&new\_expr, 0, \mathbf{sizeof}(\mathbf{mp\_value}));$
                $new\_number(new\_expr.data.n);$
                $mp\_disp\_err(mp, \Lambda);$
                $set\_number\_to\_zero(new\_expr.data.n);$
                $mp\_back\_error(mp, \texttt{"Improper\textvisiblespace `addto'"}, hlp, true);$
                $mp\_get\_x\_next(mp);$
                $mp\_flush\_cur\_exp(mp, new\_expr);$
            $\}$

```
        else if (add_type ≡ contour_code) {
          if (mp_left_type(cur_exp_knot()) ≡ mp_endpoint) {      /* Complain about a non-cycle */
            const char *hlp[] = {"That␣contour␣should␣have␣ended␣with␣`..cycle'␣or␣`&cycle'.",
                "So␣I'll␣not␣change␣anything␣just␣now.", Λ};

            mp_back_error(mp, "Not␣a␣cycle", hlp, true);
            mp_get_x_next(mp);
          }
          else {
            p = mp_new_fill_node(mp, cur_exp_knot());
            mp→cur_exp.type = mp_vacuous;
          }
        }
        else {
          p = mp_new_stroked_node(mp, cur_exp_knot());
          mp→cur_exp.type = mp_vacuous;
        }
      }
      mp_scan_with_list(mp, p);      /* Use p, e, and add_type to augment lhv as requested */
      lhe = mp_find_edges_var(mp, lhv);
      if (lhe ≡ Λ) {
        if ((e ≡ Λ) ∧ (p ≠ Λ))  e = mp_toss_gr_object(mp, p);
        if (e ≠ Λ)  delete_edge_ref(e);
      }
      else if (add_type ≡ also_code) {
        if (e ≠ Λ) {      /* Merge e into lhe and delete e */
          if (mp_link(edge_list(e)) ≠ Λ) {
            mp_link(obj_tail(lhe)) = mp_link(edge_list(e));
            obj_tail(lhe) = obj_tail(e);
            obj_tail(e) = edge_list(e);
            mp_link(edge_list(e)) = Λ;
            mp_flush_dash_list(mp, lhe);
          }
          mp_toss_edges(mp, e);
        }
      }
      else if (p ≠ Λ) {
        mp_link(obj_tail(lhe)) = p;
        obj_tail(lhe) = p;
        if (add_type ≡ double_path_code) {
          if (mp_pen_p((mp_stroked_node) p) ≡ Λ) {
            mp_pen_p((mp_stroked_node) p) = mp_get_pen_circle(mp, zero_t);
          }
        }
      }
    }
  }
}
```

1137. ⟨ Declare action procedures for use by *do_statement* 1048 ⟩ +≡
  ⟨ Declare the PostScript output procedures 1269 ⟩;

  static void *mp_do_ship_out*(**MP** *mp*);

1138.   **void** $mp\_do\_ship\_out(\mathbf{MP}\ mp)$
  {
      integer c;   /∗ the character code ∗/
      mp_value new_expr;

      $memset(\&new\_expr, 0, \mathbf{sizeof}(\mathbf{mp\_value}));$
      $new\_number(new\_expr.data.n);$
      $mp\_get\_x\_next(mp);$
      $mp\_scan\_expression(mp);$
      if $(mp \rightarrow cur\_exp.type \neq mp\_picture\_type)$ {
          ⟨ Complain that it's not a known picture 1139 ⟩;
      }
      else {
          $c = round\_unscaled(internal\_value(mp\_char\_code))\ \%\ 256;$
          if $(c < 0)\ \ c = c + 256;$
          ⟨ Store the width information for character code c 1173 ⟩;
          $mp\_ship\_out(mp, cur\_exp\_node(\ ));$
          $set\_number\_to\_zero(new\_expr.data.n);$
          $mp\_flush\_cur\_exp(mp, new\_expr);$
      }
  }

1139.   ⟨ Complain that it's not a known picture 1139 ⟩ ≡
  {
      const char $*hlp[\ ] = \{$"I␣can␣only␣output␣known␣pictures.", $\Lambda\};$

      $mp\_disp\_err(mp, \Lambda);$
      $set\_number\_to\_zero(new\_expr.data.n);$
      $mp\_back\_error(mp,$ "Not␣a␣known␣picture", $hlp, true);$
      $mp\_get\_x\_next(mp);$
      $mp\_flush\_cur\_exp(mp, new\_expr);$
  }
This code is used in section 1138.

1140.   The **everyjob** command simply assigns a nonzero value to the global variable $start\_sym$.

1141.   ⟨ Global variables 14 ⟩ +≡
  mp_sym $start\_sym$;   /∗ a symbolic token to insert at beginning of job ∗/

1142.   ⟨ Set initial values of key variables 38 ⟩ +≡
  $mp \rightarrow start\_sym = \Lambda;$

1143.   Finally, we have only the "message" commands remaining.

#define *message_code*   0
#define *err_message_code*   1
#define *err_help_code*   2
#define *filename_template_code*   3
#define *print_with_leading_zeroes*(A, B) **do**
       {
          size_t $g = mp\text{-}cur\_length$;
          size_t $f = (\textbf{size\_t})(B)$;
          mp_print_int$(mp, (A))$;
          $g = mp\text{-}cur\_length - g$;
          if $(f > g)$ {
             $mp\text{-}cur\_length = mp\text{-}cur\_length - g$;
             while $(f > g)$ {
                mp_print_char$(mp, xord(\texttt{'0'}))$;
                decr(f);
             }
             ;
             mp_print_int$(mp, (A))$;
          }
          ;
          $f = 0$;
       }
       while (0)

⟨ Put each of METAPOST's primitives into the hash table 200 ⟩ +≡
  mp_primitive$(mp, \texttt{"message"}, mp\_message\_command, message\_code)$;
  ;
  mp_primitive$(mp, \texttt{"errmessage"}, mp\_message\_command, err\_message\_code)$;
  ;
  mp_primitive$(mp, \texttt{"errhelp"}, mp\_message\_command, err\_help\_code)$;
  ;
  mp_primitive$(mp, \texttt{"filenametemplate"}, mp\_message\_command, filename\_template\_code)$;

1144.   ⟨ Cases of *print_cmd_mod* for symbolic printing of primitives 233 ⟩ +≡
case *mp_message_command*:
  if $(m < err\_message\_code)$ *mp_print*$(mp, \texttt{"message"})$;
  else if $(m \equiv err\_message\_code)$ *mp_print*$(mp, \texttt{"errmessage"})$;
  else if $(m \equiv filename\_template\_code)$ *mp_print*$(mp, \texttt{"filenametemplate"})$;
  else *mp_print*$(mp, \texttt{"errhelp"})$;
  break;

1145.   ⟨ Declare action procedures for use by *do_statement* 1048 ⟩ +≡
  ⟨ Declare a procedure called *no_string_err* 1148 ⟩;
  static void *mp_do_message*(**MP** *mp*);

1146.

```
void mp_do_message(MP mp)
{
    int m;     /* the type of message */
    mp_value new_expr;
    m = cur_mod();
    memset(&new_expr, 0, sizeof(mp_value));
    new_number(new_expr.data.n);
    mp_get_x_next(mp);
    mp_scan_expression(mp);
    if (mp→cur_exp.type ≠ mp_string_type)
        mp_no_string_err(mp, "A␣message␣should␣be␣a␣known␣string␣expression.");
    else {
        switch (m) {
        case message_code: mp_print_nl(mp, "");
            mp_print_str(mp, cur_exp_str());
            break;
        case err_message_code: ⟨Print string cur_exp as an error message 1152⟩;
            break;
        case err_help_code: ⟨Save string cur_exp as the err_help 1149⟩;
            break;
        case filename_template_code: ⟨Save the filename template 1147⟩;
            break;
        }     /* there are no other cases */
    }
    set_number_to_zero(new_expr.data.n);
    mp_flush_cur_exp(mp, new_expr);
}
```

1147.   ⟨Save the filename template 1147⟩ ≡

```
{
    delete_str_ref(internal_string(mp_output_template));
    if (cur_exp_str()→len ≡ 0) {
        set_internal_string(mp_output_template, mp_rts(mp, "%j.%c"));
    }
    else {
        set_internal_string(mp_output_template, cur_exp_str());
        add_str_ref(internal_string(mp_output_template));
    }
}
```

This code is used in section 1146.

1148.   ⟨Declare a procedure called *no_string_err* 1148⟩ ≡
　　static void *mp_no_string_err*(**MP** *mp*, **const char** *∗s*)
　　{
　　　　const char *∗hlp*[ ] = {*s*, Λ};
　　　　mp_disp_err(*mp*, Λ);
　　　　mp_back_error(*mp*, "Not␣a␣string", *hlp*, *true*);
　　　　;
　　　　mp_get_x_next(*mp*);
　　}

This code is used in section 1145.

1149.   The global variable *err_help* is zero when the user has most recently given an empty help string, or if none has ever been given.

⟨Save string *cur_exp* as the *err_help* 1149⟩ ≡
　　{
　　　　if (*mp*→*err_help* ≠ Λ) *delete_str_ref*(*mp*→*err_help*);
　　　　if (*cur_exp_str*( )→*len* ≡ 0) *mp*→*err_help* = Λ;
　　　　else {
　　　　　　mp→*err_help* = *cur_exp_str*( );
　　　　　　add_str_ref(*mp*→*err_help*);
　　　　}
　　}

This code is used in section 1146.

1150.   If **errmessage** occurs often in *mp_scroll_mode*, without user-defined **errhelp**, we don't want to give a long help message each time. So we give a verbose explanation only once.

⟨Global variables 14⟩ +≡
　　boolean *long_help_seen*;　　/∗ has the long \errmessage help been used? ∗/

1151.   ⟨Set initial values of key variables 38⟩ +≡
　　mp→*long_help_seen* = *false*;

1152.   ⟨Print string *cur_exp* as an error message 1152⟩ ≡

  {

    char *msg*[256];

    mp_snprintf (*msg*, 256, "%s", *mp_str* (*mp*, *cur_exp_str* ( )));

    if (*mp*→*err_help* ≠ Λ) {

      mp→*use_err_help* = *true*;

      mp_back_error (*mp*, *msg*, Λ, *true*);

    }

    else if (*mp*→*long_help_seen*) {

      const char *∗hlp*[ ] = {"(That␣was␣another␣'errmessage'.)", Λ};

      mp_back_error (*mp*, *msg*, *hlp*, *true*);

    }

    else {

      const char *∗hlp*[ ] = {"This␣error␣message␣was␣generated␣by␣an␣'errmessage'",

        "command,␣so␣I␣can\'t␣give␣any␣explicit␣help.",

        "Pretend␣that␣you're␣Miss␣Marple:␣Examine␣all␣clues,",

        "and␣deduce␣the␣truth␣by␣inspired␣guesses.", Λ};

      if (*mp*→*interaction* < *mp_error_stop_mode*) *mp*→*long_help_seen* = *true*;

      mp_back_error (*mp*, *msg*, *hlp*, *true*);

    }

    mp_get_x_next (*mp*);

    mp→*use_err_help* = *false*;

  }

This code is used in section 1146.

1153.   ⟨Declare action procedures for use by *do_statement* 1048⟩ +≡

  static void *mp_do_write* (**MP** *mp*);

1154.   **void** mp_do_write(**MP** mp)
  {
    mp_string t;   /∗ the line of text to be written ∗/
    write_index n, $n0$;   /∗ for searching wr_fname and wr_file arrays ∗/
    unsigned $old\_setting$;   /∗ for saving $selector$ during output ∗/
    mp_value new_expr;
    $memset(\&new\_expr, 0, \textbf{sizeof}(\textbf{mp\_value}))$;
    $new\_number(new\_expr.data.n)$;
    $mp\_get\_x\_next(mp)$;
    $mp\_scan\_expression(mp)$;
    if $(mp\text{→}cur\_exp.type \neq mp\_string\_type)$ {
      $mp\_no\_string\_err(mp, "The␣text␣to␣be␣written␣should␣be␣a␣known␣string␣expression")$;
    }
    else if $(cur\_cmd() \neq mp\_to\_token)$ {
      const char $*hlp[\,] = \{"A␣write␣command␣should␣end␣with␣`to␣<filename>`", \Lambda\}$;

      $mp\_back\_error(mp, "Missing␣`to`␣clause", hlp, true)$;
      $mp\_get\_x\_next(mp)$;
    }
    else {
      $t = cur\_exp\_str()$;
      $mp\text{→}cur\_exp.type = mp\_vacuous$;
      $mp\_get\_x\_next(mp)$;
      $mp\_scan\_expression(mp)$;
      if $(mp\text{→}cur\_exp.type \neq mp\_string\_type)$
        $mp\_no\_string\_err(mp, "I␣can\'t␣write␣to␣that␣file␣name.␣␣It␣isn't␣a␣known␣string")$;
      else {
        ⟨Write t to the file named by cur_exp 1155⟩;
      }   /∗ $delete\_str\_ref(t)$; ∗/   /∗ todo: is this right? ∗/
    }
    $set\_number\_to\_zero(new\_expr.data.n)$;
    $mp\_flush\_cur\_exp(mp, new\_expr)$;
  }

1155.   ⟨Write t to the file named by cur_exp 1155⟩ ≡
  {
    ⟨Find n where $wr\_fname[n] = cur\_exp$ and call $open\_write\_file$ if cur_exp must be inserted 1156⟩;
    if $(mp\_str\_vs\_str(mp, t, mp\text{→}eof\_line) \equiv 0)$ {
      ⟨Record the end of file on $wr\_file[n]$ 1157⟩;
    }
    else {
      $old\_setting = mp\text{→}selector$;
      $mp\text{→}selector = n + write\_file$;
      $mp\_print\_str(mp, t)$;
      $mp\_print\_ln(mp)$;
      $mp\text{→}selector = old\_setting$;
    }
  }
This code is used in section 1154.

1156.   ⟨Find n where $wr\_fname[n] = cur\_exp$ and call $open\_write\_file$ if cur_exp must be inserted 1156⟩ ≡

```
{
    char *fn = mp_str(mp, cur_exp_str());
    n = mp→write_files;
    n0 = mp→write_files;
    while (mp_xstrcmp(fn, mp→wr_fname[n]) ≠ 0) {
        if (n ≡ 0) {      /* bottom reached */
            if (n0 ≡ mp→write_files) {
                if (mp→write_files < mp→max_write_files) {
                    incr(mp→write_files);
                }
                else {
                    void **wr_file;
                    char **wr_fname;
                    write_index l, k;

                    l = mp→max_write_files + (mp→max_write_files/4);
                    wr_file = xmalloc((l + 1), sizeof(void *));
                    wr_fname = xmalloc((l + 1), sizeof(char *));
                    for (k = 0; k ≤ l; k++) {
                        if (k ≤ mp→max_write_files) {
                            wr_file[k] = mp→wr_file[k];
                            wr_fname[k] = mp→wr_fname[k];
                        }
                        else {
                            wr_file[k] = 0;
                            wr_fname[k] = Λ;
                        }
                    }
                    xfree(mp→wr_file);
                    xfree(mp→wr_fname);
                    mp→max_write_files = l;
                    mp→wr_file = wr_file;
                    mp→wr_fname = wr_fname;
                }
            }
            n = n0;
            mp_open_write_file(mp, fn, n);
        }
        else {
            decr(n);
            if (mp→wr_fname[n] ≡ Λ) n0 = n;
        }
    }
}
```

This code is used in section 1155.

1157. ⟨ Record the end of file on *wr_file*[*n*] 1157 ⟩ ≡

    {
      (*mp*→*close_file*)(*mp*, *mp*→*wr_file*[*n*]);
      xfree(*mp*→*wr_fname*[*n*]);
      if (*n* ≡ *mp*→*write_files* − 1) *mp*→*write_files* = *n*;
    }

This code is used in section 1155.

1158.   Writing font metric data.   TₑX gets its knowledge about fonts from font metric files, also called TFM files; the 'T' in 'TFM' stands for TₑX, but other programs know about them too. One of METAPOST's duties is to write TFM files so that the user's fonts can readily be applied to typesetting.

The information in a TFM file appears in a sequence of 8-bit bytes. Since the number of bytes is always a multiple of 4, we could also regard the file as a sequence of 32-bit words, but METAPOST uses the byte interpretation. The format of TFM files was designed by Lyle Ramshaw in 1980. The intent is to convey a lot of different kinds of information in a compact but useful form.

⟨ Global variables 14 ⟩ +≡
  void *tfm_file*;   /∗ the font metric output goes here ∗/
  char *metric_file_name*;   /∗ full name of the font metric file ∗/

1159.   The first 24 bytes (6 words) of a TFM file contain twelve 16-bit integers that give the lengths of the various subsequent portions of the file. These twelve integers are, in order:

$$
\begin{aligned}
lf &= \text{length of the entire file, in words;} \\
lh &= \text{length of the header data, in words;} \\
bc &= \text{smallest character code in the font;} \\
ec &= \text{largest character code in the font;} \\
nw &= \text{number of words in the width table;} \\
nh &= \text{number of words in the height table;} \\
nd &= \text{number of words in the depth table;} \\
ni &= \text{number of words in the italic correction table;} \\
nl &= \text{number of words in the lig/kern table;} \\
nk &= \text{number of words in the kern table;} \\
ne &= \text{number of words in the extensible character table;} \\
np &= \text{number of font parameter words.}
\end{aligned}
$$

They are all nonnegative and less than 2^{15}. We must have $bc - 1 \le ec \le 255$, $ne \le 256$, and

$$lf = 6 + lh + (ec - bc + 1) + nw + nh + nd + ni + nl + nk + ne + np.$$

Note that a font may contain as many as 256 characters (if $bc = 0$ and $ec = 255$), and as few as 0 characters (if $bc = ec + 1$).

Incidentally, when two or more 8-bit bytes are combined to form an integer of 16 or more bits, the most significant bytes appear first in the file. This is called BigEndian order.

1160.   The rest of the TFM file may be regarded as a sequence of ten data arrays.

The most important data type used here is a *fix_word*, which is a 32-bit representation of a binary fraction. A *fix_word* is a signed quantity, with the two's complement of the entire word used to represent negation. Of the 32 bits in a *fix_word*, exactly 12 are to the left of the binary point; thus, the largest *fix_word* value is $2048 - 2^{-20}$, and the smallest is -2048. We will see below, however, that all but two of the *fix_word* values must lie between -16 and $+16$.

1161.   The first data array is a block of header information, which contains general facts about the font. The header must contain at least two words, $header[0]$ and $header[1]$, whose meaning is explained below. Additional header information of use to other software routines might also be included, and METAPOST will generate it if the `headerbyte` command occurs. For example, 16 more words of header information are in use at the Xerox Palo Alto Research Center; the first ten specify the character coding scheme used (e.g., '`XEROX TEXT`' or '`TEX MATHSY`'), the next five give the font family name (e.g., '`HELVETICA`' or '`CMSY`'), and the last gives the "face byte."

$header[0]$ is a 32-bit check sum that METAPOST will copy into the `GF` output file. This helps ensure consistency between files, since TEX records the check sums from the `TFM`'s it reads, and these should match the check sums on actual fonts that are used. The actual relation between this check sum and the rest of the `TFM` file is not important; the check sum is simply an identification number with the property that incompatible fonts almost always have distinct check sums.

$header[1]$ is a fix_word containing the design size of the font, in units of TEX points. This number must be at least 1.0; it is fairly arbitrary, but usually the design size is 10.0 for a "10 point" font, i.e., a font that was designed to look best at a 10-point size, whatever that really means. When a TEX user asks for a font 'at δ `pt`', the effect is to override the design size and replace it by δ, and to multiply the x and y coordinates of the points in the font image by a factor of δ divided by the design size. *All other dimensions in the `TFM` file are fix_word numbers in design-size units.* Thus, for example, the value of $param[6]$, which defines the `em` unit, is often the fix_word value $2^{20} = 1.0$, since many fonts have a design size equal to one em. The other dimensions must be less than 16 design-size units in absolute value; thus, $header[1]$ and $param[1]$ are the only fix_word entries in the whole `TFM` file whose first byte might be something besides 0 or 255.

1162.   Next comes the $char\_info$ array, which contains one $char\_info\_word$ per character. Each word in this part of the file contains six fields packed into four bytes as follows.

first byte: $width\_index$ (8 bits)
second byte: $height\_index$ (4 bits) times 16, plus $depth\_index$ (4 bits)
third byte: $italic\_index$ (6 bits) times 4, plus tag (2 bits)
fourth byte: $remainder$ (8 bits)

The actual width of a character is $width[width\_index]$, in design-size units; this is a device for compressing information, since many characters have the same width. Since it is quite common for many characters to have the same height, depth, or italic correction, the `TFM` format imposes a limit of 16 different heights, 16 different depths, and 64 different italic corrections.

Incidentally, the relation $width[0] = height[0] = depth[0] = italic[0] = 0$ should always hold, so that an index of zero implies a value of zero. The $width\_index$ should never be zero unless the character does not exist in the font, since a character is valid if and only if it lies between bc and ec and has a nonzero $width\_index$.

1163.   The *tag* field in a *char_info_word* has four values that explain how to interpret the *remainder* field.

$tag = 0$ (*no_tag*) means that *remainder* is unused.

$tag = 1$ (*lig_tag*) means that this character has a ligature/kerning program starting at location *remainder* in the *lig_kern* array.

$tag = 2$ (*list_tag*) means that this character is part of a chain of characters of ascending sizes, and not the largest in the chain. The *remainder* field gives the character code of the next larger character.

$tag = 3$ (*ext_tag*) means that this character code represents an extensible character, i.e., a character that is built up of smaller pieces so that it can be made arbitrarily large. The pieces are specified in *exten*[*remainder*].

Characters with $tag = 2$ and $tag = 3$ are treated as characters with $tag = 0$ unless they are used in special circumstances in math formulas. For example, TeX's \sum operation looks for a *list_tag*, and the \left operation looks for both *list_tag* and *ext_tag*.

#define   *no_tag*   0   /* vanilla character */
#define   *lig_tag*   1   /* character has a ligature/kerning program */
#define   *list_tag*   2   /* character has a successor in a charlist */
#define   *ext_tag*   3   /* character is extensible */

1164. The *lig_kern* array contains instructions in a simple programming language that explains what to do for special letter pairs. Each word in this array is a *lig_kern_command* of four bytes.

first byte: *skip_byte*, indicates that this is the final program step if the byte is 128 or more, otherwise the next step is obtained by skipping this number of intervening steps.

second byte: *next_char*, "if *next_char* follows the current character, then perform the operation and stop, otherwise continue."

third byte: *op_byte*, indicates a ligature step if less than 128, a kern step otherwise.

fourth byte: *remainder*.

In a kern step, an additional space equal to $kern[256 * (op\_byte - 128) + remainder]$ is inserted between the current character and *next_char*. This amount is often negative, so that the characters are brought closer together by kerning; but it might be positive.

There are eight kinds of ligature steps, having *op_byte* codes $4a+2b+c$ where $0 \le a \le b+c$ and $0 \le b, c \le 1$. The character whose code is *remainder* is inserted between the current character and *next_char*; then the current character is deleted if $b = 0$, and *next_char* is deleted if $c = 0$; then we pass over a characters to reach the next current character (which may have a ligature/kerning program of its own).

If the very first instruction of the *lig_kern* array has *skip_byte* $= 255$, the *next_char* byte is the so-called right boundary character of this font; the value of *next_char* need not lie between *bc* and *ec*. If the very last instruction of the *lig_kern* array has *skip_byte* $= 255$, there is a special ligature/kerning program for a left boundary character, beginning at location $256 * op\_byte + remainder$. The interpretation is that TEX puts implicit boundary characters before and after each consecutive string of characters from the same font. These implicit characters do not appear in the output, but they can affect ligatures and kerning.

If the very first instruction of a character's *lig_kern* program has *skip_byte* > 128, the program actually begins in location $256 * op\_byte + remainder$. This feature allows access to large *lig_kern* arrays, because the first instruction must otherwise appear in a location ≤ 255.

Any instruction with *skip_byte* > 128 in the *lig_kern* array must satisfy the condition

$$256 * op\_byte + remainder < nl.$$

If such an instruction is encountered during normal program execution, it denotes an unconditional halt; no ligature command is performed.

#define *stop_flag* (128)   /∗ value indicating 'STOP' in a lig/kern program ∗/
#define *kern_flag* (128)   /∗ op code for a kern step ∗/
#define *skip_byte*(A)   *mp→lig_kern*[(A)].*b0*
#define *next_char*(A)   *mp→lig_kern*[(A)].*b1*
#define *op_byte*(A)   *mp→lig_kern*[(A)].*b2*
#define *rem_byte*(A)   *mp→lig_kern*[(A)].*b3*

1165. Extensible characters are specified by an *extensible_recipe*, which consists of four bytes called *top*, *mid*, *bot*, and *rep* (in this order). These bytes are the character codes of individual pieces used to build up a large symbol. If *top*, *mid*, or *bot* are zero, they are not present in the built-up result. For example, an extensible vertical line is like an extensible bracket, except that the top and bottom pieces are missing.

Let T, M, B, and R denote the respective pieces, or an empty box if the piece isn't present. Then the extensible characters have the form $TR^k M R^k B$ from top to bottom, for some $k \ge 0$, unless M is absent; in the latter case we can have $TR^k B$ for both even and odd values of k. The width of the extensible character is the width of R; and the height-plus-depth is the sum of the individual height-plus-depths of the components used, since the pieces are butted together in a vertical list.

#define *ext_top*(A)   *mp→exten*[(A)].*b0*   /∗ *top* piece in a recipe ∗/
#define *ext_mid*(A)   *mp→exten*[(A)].*b1*   /∗ *mid* piece in a recipe ∗/
#define *ext_bot*(A)   *mp→exten*[(A)].*b2*   /∗ *bot* piece in a recipe ∗/
#define *ext_rep*(A)   *mp→exten*[(A)].*b3*   /∗ *rep* piece in a recipe ∗/

1166.   The final portion of a TFM file is the *param* array, which is another sequence of *fix_word* values.

param[1] = *slant* is the amount of italic slant, which is used to help position accents. For example, *slant* = .25 means that when you go up one unit, you also go .25 units to the right. The *slant* is a pure number; it is the only *fix_word* other than the design size itself that is not scaled by the design size.

param[2] = *space* is the normal spacing between words in text. Note that character 040 in the font need not have anything to do with blank spaces.

param[3] = *space_stretch* is the amount of glue stretching between words.

param[4] = *space_shrink* is the amount of glue shrinking between words.

param[5] = *x_height* is the size of one ex in the font; it is also the height of letters for which accents don't have to be raised or lowered.

param[6] = *quad* is the size of one em in the font.

param[7] = *extra_space* is the amount added to *param*[2] at the ends of sentences.

If fewer than seven parameters are present, TeX sets the missing parameters to zero.

#define   *slant_code*   1
#define   *space_code*   2
#define   *space_stretch_code*   3
#define   *space_shrink_code*   4
#define   *x_height_code*   5
#define   *quad_code*   6
#define   *extra_space_code*   7

1167.   So that is what TFM files hold. One of METAPOST's duties is to output such information, and it does this all at once at the end of a job. In order to prepare for such frenetic activity, it squirrels away the necessary facts in various arrays as information becomes available.

Character dimensions (**charwd**, **charht**, **chardp**, and **charic**) are stored respectively in *tfm_width*, *tfm_height*, *tfm_depth*, and *tfm_ital_corr*. Other information about a character (e.g., about its ligatures or successors) is accessible via the *char_tag* and *char_remainder* arrays. Other information about the font as a whole is kept in additional arrays called *header_byte*, *lig_kern*, *kern*, *exten*, and *param*.

#define   *max_tfm_int*   32510
#define   *undefined_label*   *max_tfm_int*   /∗ an undefined local label ∗/

⟨ Global variables 14 ⟩ +≡
#define TFM_ITEMS   257
 eight_bits *bc*;
 eight_bits *ec*;   /∗ smallest and largest character codes shipped out ∗/
 mp_node *tfm_width*[TFM_ITEMS];   /∗ **charwd** values ∗/
 mp_node *tfm_height*[TFM_ITEMS];   /∗ **charht** values ∗/
 mp_node *tfm_depth*[TFM_ITEMS];   /∗ **chardp** values ∗/
 mp_node *tfm_ital_corr*[TFM_ITEMS];   /∗ **charic** values ∗/
 boolean *char_exists*[TFM_ITEMS];   /∗ has this code been shipped out? ∗/
 int *char_tag*[TFM_ITEMS];   /∗ *remainder* category ∗/
 int *char_remainder*[TFM_ITEMS];   /∗ the *remainder* byte ∗/
 char ∗*header_byte*;   /∗ bytes of the TFM header ∗/
 int *header_last*;   /∗ last initialized TFM header byte ∗/
 int *header_size*;   /∗ size of the TFM header ∗/
 four_quarters ∗*lig_kern*;   /∗ the ligature/kern table ∗/
 short *nl*;   /∗ the number of ligature/kern steps so far ∗/
 mp_number ∗*kern*;   /∗ distinct kerning amounts ∗/
 short *nk*;   /∗ the number of distinct kerns so far ∗/
 four_quarters *exten*[TFM_ITEMS];   /∗ extensible character recipes ∗/
 short *ne*;   /∗ the number of extensible characters so far ∗/
 mp_number ∗*param*;   /∗ **fontinfo** parameters ∗/
 short *np*;   /∗ the largest **fontinfo** parameter specified so far ∗/
 short *nw*;
 short *nh*;
 short *nd*;
 short *ni*;   /∗ sizes of TFM subtables ∗/
 short *skip_table*[TFM_ITEMS];   /∗ local label status ∗/
 boolean *lk_started*;   /∗ has there been a lig/kern step in this command yet? ∗/
 integer *bchar*;   /∗ right boundary character ∗/
 short *bch_label*;   /∗ left boundary starting location ∗/
 short *ll*;
 short *lll*;   /∗ registers used for lig/kern processing ∗/
 short *label_loc*[257];   /∗ lig/kern starting addresses ∗/
 eight_bits *label_char*[257];   /∗ characters for *label_loc* ∗/
 short *label_ptr*;   /∗ highest position occupied in *label_loc* ∗/

1168.   ⟨ Allocate or initialize variables 28 ⟩ +≡
 mp→*header_last* = 7;
 mp→*header_size* = 128;   /∗ just for init ∗/
 mp→*header_byte* = *xmalloc*(*mp*→*header_size*, **sizeof**(**char**));

1169. ⟨Dealloc variables 27⟩ +≡
  $xfree(mp\rightarrow header\_byte)$;
  $xfree(mp\rightarrow lig\_kern)$;
  if $(mp\rightarrow kern)$ {
    int i;
    for $(i = 0;\ i < (max\_tfm\_int + 1);\ i{+}{+})$ {
      $free\_number(mp\rightarrow kern[i])$;
    }
    $xfree(mp\rightarrow kern)$;
  }
  if $(mp\rightarrow param)$ {
    int i;
    for $(i = 0;\ i < (max\_tfm\_int + 1);\ i{+}{+})$ {
      $free\_number(mp\rightarrow param[i])$;
    }
    $xfree(mp\rightarrow param)$;
  }

1170. ⟨Set initial values of key variables 38⟩ +≡
  for $(k = 0;\ k \le 255;\ k{+}{+})$ {
    $mp\rightarrow tfm\_width[k] = 0$;
    $mp\rightarrow tfm\_height[k] = 0$;
    $mp\rightarrow tfm\_depth[k] = 0$;
    $mp\rightarrow tfm\_ital\_corr[k] = 0$;
    $mp\rightarrow char\_exists[k] = false$;
    $mp\rightarrow char\_tag[k] = no\_tag$;
    $mp\rightarrow char\_remainder[k] = 0$;
    $mp\rightarrow skip\_table[k] = undefined\_label$;
  }
  $memset(mp\rightarrow header\_byte, 0, (\textbf{size\_t})\ mp\rightarrow header\_size)$;
  $mp\rightarrow bc = 255$;
  $mp\rightarrow ec = 0$;
  $mp\rightarrow nl = 0$;
  $mp\rightarrow nk = 0$;
  $mp\rightarrow ne = 0$;
  $mp\rightarrow np = 0$;
  $set\_internal\_from\_number(mp\_boundary\_char, unity\_t)$;
  $number\_negate(internal\_value(mp\_boundary\_char))$;
  $mp\rightarrow bch\_label = undefined\_label$;
  $mp\rightarrow label\_loc[0] = -1$;
  $mp\rightarrow label\_ptr = 0$;

1171. ⟨Declarations 8⟩ +≡
  static mp_node $mp\_tfm\_check(\textbf{MP}\ mp, \textbf{quarterword}\ m)$;

1172.   **static mp_node** *mp_tfm_check*(**MP** *mp*, **quarterword** *m*)
{
    mp_number *absm*;
    mp_node *p* = *mp_get_value_node*(*mp*);
    new_number(*absm*);
    number_clone(*absm*, *internal_value*(*m*));
    number_abs(*absm*);
    if (*number_greaterequal*(*absm*, *fraction_half_t*)) {
      char *msg*[256];
      const char *∗hlp*[ ] = {"Font␣metric␣dimensions␣must␣be␣less␣than␣2048pt.", Λ};
      mp_snprintf(*msg*, 256, "Enormous␣%s␣has␣been␣reduced", *internal_name*(*m*));
      ;
      mp_back_error(*mp*, *msg*, *hlp*, *true*);
      mp_get_x_next(*mp*);
      if (*number_positive*(*internal_value*(*m*))) {
        set_value_number(*p*, *fraction_half_t*);
        number_add_scaled(*value_number*(*p*), −1);
      }
      else {
        set_value_number(*p*, *fraction_half_t*);
        number_negate(*value_number*(*p*));
        number_add_scaled(*value_number*(*p*), 1);
      }
    }
    else {
      set_value_number(*p*, *internal_value*(*m*));
    }
    free_number(*absm*);
    return *p*;
}

1173.   ⟨ Store the width information for character code *c* 1173 ⟩ ≡
  if (*c* < *mp*→*bc*) *mp*→*bc* = (**eight_bits**) *c*;
  if (*c* > *mp*→*ec*) *mp*→*ec* = (**eight_bits**) *c*;
  mp→*char_exists*[*c*] = *true*;
  mp_free_value_node(*mp*, *mp*→*tfm_width*[*c*]);
  mp→*tfm_width*[*c*] = *mp_tfm_check*(*mp*, *mp_char_wd*);
  mp_free_value_node(*mp*, *mp*→*tfm_height*[*c*]);
  mp→*tfm_height*[*c*] = *mp_tfm_check*(*mp*, *mp_char_ht*);
  mp_free_value_node(*mp*, *mp*→*tfm_depth*[*c*]);
  mp→*tfm_depth*[*c*] = *mp_tfm_check*(*mp*, *mp_char_dp*);
  mp_free_value_node(*mp*, *mp*→*tfm_ital_corr*[*c*]); *mp*→*tfm_ital_corr*[*c*] = *mp_tfm_check*(*mp*, *mp_char_ic*)
This code is used in section 1138.

1174.   Now let's consider METAPOST's special TFM-oriented commands.

1175.   #**define** *char_list_code*   0
#**define** *lig_table_code*   1
#**define** *extensible_code*   2
#**define** *header_byte_code*   3
#**define** *font_dimen_code*   4

⟨ Put each of METAPOST's primitives into the hash table 200 ⟩ +≡
  mp_primitive(*mp*, "charlist", *mp_tfm_command*, *char_list_code*);
  ;
  mp_primitive(*mp*, "ligtable", *mp_tfm_command*, *lig_table_code*);
  ;
  mp_primitive(*mp*, "extensible", *mp_tfm_command*, *extensible_code*);
  ;
  mp_primitive(*mp*, "headerbyte", *mp_tfm_command*, *header_byte_code*);
  ;
  mp_primitive(*mp*, "fontdimen", *mp_tfm_command*, *font_dimen_code*);

1176.   ⟨ Cases of *print_cmd_mod* for symbolic printing of primitives 233 ⟩ +≡
case *mp_tfm_command*:
  switch (*m*) {
  case *char_list_code*: *mp_print*(*mp*, "charlist");
    break;
  case *lig_table_code*: *mp_print*(*mp*, "ligtable");
    break;
  case *extensible_code*: *mp_print*(*mp*, "extensible");
    break;
  case *header_byte_code*: *mp_print*(*mp*, "headerbyte");
    break;
  default: *mp_print*(*mp*, "fontdimen");
    break;
  }
  break;

1177.   ⟨ Declare action procedures for use by *do_statement* 1048 ⟩ +≡
  static eight_bits *mp_get_code*(**MP** *mp*);

1178.   **eight_bits** mp_get_code(**MP** mp)
{   /∗ scans a character code value ∗/
  integer c;   /∗ the code value found ∗/
  mp_value new_expr;
  const char ∗$hlp[\,] = \{$"I␣was␣looking␣for␣a␣number␣between␣0␣and␣255,␣or␣for␣a",
      "string␣of␣length␣1.␣Didn't␣find␣it;␣will␣use␣0␣instead.", $\Lambda\}$;
  $memset(\&new\_expr, 0, \mathbf{sizeof}(\mathbf{mp\_value}))$;
  $new\_number(new\_expr.data.n)$;
  $mp\_get\_x\_next(mp)$;
  $mp\_scan\_expression(mp)$;
  if $(mp\text{→}cur\_exp.type \equiv mp\_known)$ {
    $c = round\_unscaled(cur\_exp\_value\_number(\,))$;
    if $(c \geq 0)$
      if $(c < 256)$ **return** (**eight_bits**) c;
  }
  else if $(mp\text{→}cur\_exp.type \equiv mp\_string\_type)$ {
    if $(cur\_exp\_str(\,)\text{→}len \equiv 1)$ {
      $c = (\mathbf{integer})(∗(cur\_exp\_str(\,)\text{→}str))$;
      return (**eight_bits**) c;
    }
  }
  $mp\_disp\_err(mp, \Lambda)$;
  $set\_number\_to\_zero(new\_expr.data.n)$;
  $mp_back_error(mp, $"Invalid␣code␣has␣been␣replaced␣by␣0"$, hlp, true)$;
  ;
  $mp\_get\_x\_next(mp)$;
  $mp\_flush\_cur\_exp(mp, new\_expr)$;
  $c = 0$;
  return (**eight_bits**) c;
}

1179.   ⟨ Declare action procedures for use by $do\_statement$ 1048 ⟩ +≡
  static void mp_set_tag(**MP** mp, **halfword** c, **quarterword** t, **halfword** r);

1180.   **void** mp_set_tag(**MP** mp, **halfword** c, **quarterword** t, **halfword** r)
  {
    if $(mp\text{→}char\_tag[c] \equiv no\_tag)$ {
      $mp\text{→}char\_tag[c] = t$;
      $mp\text{→}char\_remainder[c] = r$;
      if $(t \equiv lig\_tag)$ {
        $mp\text{→}label\_ptr\text{++}$;
        $mp\text{→}label\_loc[mp\text{→}label\_ptr] = (\mathbf{short})\ r$;
        $mp\text{→}label\_char[mp\text{→}label\_ptr] = (\mathbf{eight\_bits})\ c$;
      }
    }
    else {
      ⟨ Complain about a character tag conflict 1181 ⟩;
    }
  }

1181. ⟨Complain about a character tag conflict 1181⟩ ≡

```
{
  const char *xtra = Λ;
  char msg[256];
  const char *hlp[] = {"It's␣not␣legal␣to␣label␣a␣character␣more␣than␣once.",
      "So␣I'll␣not␣change␣anything␣just␣now.", Λ};
  switch (mp→char_tag[c]) {
  case lig_tag: xtra = "in␣a␣ligtable";
    break;
  case list_tag: xtra = "in␣a␣charlist";
    break;
  case ext_tag: xtra = "extensible";
    break;
  default: xtra = "";
    break;
  }
  if ((c > '␣') ∧ (c < 127)) {
    mp_snprintf (msg, 256, "Character␣%c␣is␣already␣%s", xord (c), xtra);
  }
  else if (c ≡ 256) {
    mp_snprintf (msg, 256, "Character␣||␣is␣already␣%s", xtra);
  }
  else {
    mp_snprintf (msg, 256, "Character␣code␣%d␣is␣already␣%s", c, xtra);
  }
  ;
  mp_back_error (mp, msg, hlp, true);
  mp_get_x_next (mp);
}
```

This code is used in section 1180.

1182. ⟨Declare action procedures for use by *do_statement* 1048⟩ +≡
  static void *mp_do_tfm_command* (**MP** *mp*);

1183.   **void** $mp\_do\_tfm\_command(\mathbf{MP}\ mp)$

{

   int $c,\ cc;$ /∗ character codes ∗/

   int $k;$ /∗ index into the $kern$ array ∗/

   int $j;$ /∗ index into $header\_byte$ or $param$ ∗/

   $\mathbf{mp\_value}\ new\_expr;$

   $memset(\&new\_expr, 0, \mathbf{sizeof}(\mathbf{mp\_value}));$

   $new\_number(new\_expr.data.n);$

   switch $(cur\_mod())$ {

   case $char\_list\_code$: $c = mp\_get\_code(mp);$ /∗ we will store a list of character successors ∗/

     while $(cur\_cmd() \equiv mp\_colon)$ {

       $cc = mp\_get\_code(mp);$

       $mp\_set\_tag(mp, c, list\_tag, cc);$

       $c = cc;$

     }

     ;

     break;

   case lig_table_code:

     if $(mp\text{→}lig\_kern \equiv \Lambda)\ mp\text{→}lig\_kern = xmalloc((max\_tfm\_int + 1), \mathbf{sizeof}(\mathbf{four\_quarters}));$

     if $(mp\text{→}kern \equiv \Lambda)$ {

       int $i;$

       $mp\text{→}kern = xmalloc((max\_tfm\_int + 1), \mathbf{sizeof}(\mathbf{mp\_number}));$

       for $(i = 0;\ i < (max\_tfm\_int + 1);\ i{+}{+})\ new\_number(mp\text{→}kern[i]);$

     }

     ⟨Store a list of ligature/kern steps 1184⟩;

     break;

   case $extensible\_code$: ⟨Define an extensible recipe 1190⟩;

     break;

   case $header\_byte\_code$: **case** $font\_dimen\_code$: $c = cur\_mod();$

    $mp\_get\_x\_next(mp);$

    $mp\_scan\_expression(mp);$

    if $((mp\text{→}cur\_exp.type \neq mp\_known) \vee number\_less(cur\_exp\_value\_number(), half\_unit\_t))$ {

      const char $*hlp[] = \{$"I␣was␣looking␣for␣a␣known,␣positive␣number.",

        "For␣safety's␣sake␣I'll␣ignore␣the␣present␣command.", $\Lambda\};$

      $mp\_disp\_err(mp, \Lambda);$

      $mp\_back\_error(mp,$ "Improper␣location"$, hlp, true);$

      ;

      $mp\_get\_x\_next(mp);$

    }

    else {

      $j = round\_unscaled(cur\_exp\_value\_number());$

      if $(cur\_cmd() \neq mp\_colon)$ {

        const char $*hlp[] = \{$"A␣colon␣should␣follow␣a␣headerbyte␣or␣fontinfo␣location.", $\Lambda\};$

        $mp\_back\_error(mp,$ "Missing␣':'␣has␣been␣inserted"$, hlp, true);$

        ;

      }

      if $(c \equiv header\_byte\_code)$ {

        ⟨Store a list of header bytes 1191⟩;

      }

      else {

        if $(mp\text{→}param \equiv \Lambda)$ {

```
                int i;

                mp→param = xmalloc((max_tfm_int + 1), sizeof(mp_number));
                for (i = 0; i < (max_tfm_int + 1); i++) new_number(mp→param[i]);
             }
             ⟨Store a list of font dimensions 1192⟩;
          }
       }
       break;
    }     /* there are no other cases */
}
```

1184.   ⟨Store a list of ligature/kern steps 1184⟩ ≡

```
  {
     mp→lk_started = false;
  CONTINUE: mp_get_x_next(mp);
     if ((cur_cmd() ≡ mp_skip_to) ∧ mp→lk_started) ⟨Process a skip_to command and goto done 1187⟩;
     if (cur_cmd() ≡ mp_bchar_label) {
        c = 256;
        set_cur_cmd((mp_variable_type)mp_colon);
     }
     else {
        mp_back_input(mp);
        c = mp_get_code(mp);
     }
     ;
     if ((cur_cmd() ≡ mp_colon) ∨ (cur_cmd() ≡ mp_double_colon)) {
        ⟨Record a label in a lig/kern subprogram and goto continue 1188⟩;
     }
     if (cur_cmd() ≡ mp_lig_kern_token) {
        ⟨Compile a ligature/kern command 1189⟩;
     }
     else {
        const char *hlp[] = {"I␣was␣looking␣for␣'=:'␣or␣'kern'␣here.", Λ};
        mp_back_error(mp, "Illegal␣ligtable␣step", hlp, true);
        ;
        next_char(mp→nl) = qi(0);
        op_byte(mp→nl) = qi(0);
        rem_byte(mp→nl) = qi(0);
        skip_byte(mp→nl) = stop_flag + 1;     /* this specifies an unconditional stop */
     }
     if (mp→nl ≡ max_tfm_int) mp_fatal_error(mp, "ligtable␣too␣large");
     mp→nl++;
     if (cur_cmd() ≡ mp_comma) goto CONTINUE;
     if (skip_byte(mp→nl − 1) < stop_flag) skip_byte(mp→nl − 1) = stop_flag;
  }
  DONE:
```
This code is used in section 1183.

1185.   ⟨Put each of METAPOST's primitives into the hash table 200⟩ +≡

  $mp\_primitive\,(mp, \text{"=:"}, mp\_lig\_kern\_token, 0)$;

  ;

  $mp\_primitive\,(mp, \text{"=:|"}, mp\_lig\_kern\_token, 1)$;

  ;

  $mp\_primitive\,(mp, \text{"=:|>"}, mp\_lig\_kern\_token, 5)$;

  ;

  $mp\_primitive\,(mp, \text{"|=:"}, mp\_lig\_kern\_token, 2)$;

  ;

  $mp\_primitive\,(mp, \text{"|=:>"}, mp\_lig\_kern\_token, 6)$;

  ;

  $mp\_primitive\,(mp, \text{"|=:|"}, mp\_lig\_kern\_token, 3)$;

  ;

  $mp\_primitive\,(mp, \text{"|=:|>"}, mp\_lig\_kern\_token, 7)$;

  ;

  $mp\_primitive\,(mp, \text{"|=:|>>"}, mp\_lig\_kern\_token, 11)$;

  ;

  $mp\_primitive\,(mp, \text{"kern"}, mp\_lig\_kern\_token, mp\_kern\_flag)$;

1186.   ⟨Cases of *print_cmd_mod* for symbolic printing of primitives 233⟩ +≡

case *mp_lig_kern_token*:

  switch (*m*) {

  case 0: $mp\_print\,(mp, \text{"=:"})$;

    break;

  case 1: $mp\_print\,(mp, \text{"=:|"})$;

    break;

  case 2: $mp\_print\,(mp, \text{"|=:"})$;

    break;

  case 3: $mp\_print\,(mp, \text{"|=:|"})$;

    break;

  case 5: $mp\_print\,(mp, \text{"=:|>"})$;

    break;

  case 6: $mp\_print\,(mp, \text{"|=:>"})$;

    break;

  case 7: $mp\_print\,(mp, \text{"|=:|>"})$;

    break;

  case 11: $mp\_print\,(mp, \text{"|=:|>>"})$;

    break;

  default: $mp\_print\,(mp, \text{"kern"})$;

    break;

  }

  break;

1187.   Local labels are implemented by maintaining the *skip_table* array, where *skip_table*[*c*] is either *undefined_label* or the address of the most recent lig/kern instruction that skips to local label *c*. In the latter case, the *skip_byte* in that instruction will (temporarily) be zero if there were no prior skips to this label, or it will be the distance to the prior skip.

We may need to cancel skips that span more than 127 lig/kern steps.

#define *cancel_skips*(*A*)   *mp*→*ll* = (*A*); **do**
    {
     mp→*lll* = *qo*(*skip_byte*(*mp*→*ll*));
     skip_byte(*mp*→*ll*) = *stop_flag*;
     mp→*ll* = (**short**)(*mp*→*ll* − *mp*→*lll*);
    }
    while (*mp*→*lll* ≠ 0)
#define *skip_error*(*A*)
    {
     const char ∗*hlp*[ ] = {"At␣most␣127␣lig/kern␣steps␣can␣separate␣skipto1␣from␣1::.", Λ};
     mp_error(*mp*, "Too␣far␣to␣skip", *hlp*, *true*);
     cancel_skips((*A*));
    }

⟨Process a *skip_to* command and **goto** *done* 1187⟩ ≡
 {
  c = *mp_get_code*(*mp*);
  if (*mp*→*nl* − *mp*→*skip_table*[*c*] > 128) {
   skip_error(*mp*→*skip_table*[*c*]);
   mp→*skip_table*[*c*] = (**short**) *undefined_label*;
  }
  if (*mp*→*skip_table*[*c*] ≡ *undefined_label*) *skip_byte*(*mp*→*nl* − 1) = *qi*(0);
  else *skip_byte*(*mp*→*nl* − 1) = *qi*(*mp*→*nl* − *mp*→*skip_table*[*c*] − 1);
  mp→*skip_table*[*c*] = (**short**)(*mp*→*nl* − 1);
  goto DONE;
 }

This code is used in section 1184.

1188.   ⟨ Record a label in a lig/kern subprogram and **goto continue** 1188 ⟩ ≡

  {

    if $(\mathit{cur\_cmd}(\,) \equiv \mathit{mp\_colon})$ {

      if $(c \equiv 256)$ $\mathit{mp}{\rightarrow}\mathit{bch\_label} = \mathit{mp}{\rightarrow}\mathit{nl}$;

      else $\mathit{mp\_set\_tag}(\mathit{mp}, c, \mathit{lig\_tag}, \mathit{mp}{\rightarrow}\mathit{nl})$;

    }

    else if $(\mathit{mp}{\rightarrow}\mathit{skip\_table}[c] < \mathit{undefined\_label})$ {

      $\mathit{mp}{\rightarrow}\mathit{ll} = \mathit{mp}{\rightarrow}\mathit{skip\_table}[c]$;

      $\mathit{mp}{\rightarrow}\mathit{skip\_table}[c] = \mathit{undefined\_label}$;

      do {

        $\mathit{mp}{\rightarrow}\mathit{lll} = \mathit{qo}(\mathit{skip\_byte}(\mathit{mp}{\rightarrow}\mathit{ll}))$;

        if $(\mathit{mp}{\rightarrow}\mathit{nl} - \mathit{mp}{\rightarrow}\mathit{ll} > 128)$ {

          $\mathit{skip\_error}(\mathit{mp}{\rightarrow}\mathit{ll})$;

          goto CONTINUE;

        }

        $\mathit{skip\_byte}(\mathit{mp}{\rightarrow}\mathit{ll}) = \mathit{qi}(\mathit{mp}{\rightarrow}\mathit{nl} - \mathit{mp}{\rightarrow}\mathit{ll} - 1)$;

        $\mathit{mp}{\rightarrow}\mathit{ll} = (\textbf{short})(\mathit{mp}{\rightarrow}\mathit{ll} - \mathit{mp}{\rightarrow}\mathit{lll})$;

      } **while** $(\mathit{mp}{\rightarrow}\mathit{lll} \neq 0)$;

    }

    goto CONTINUE;

  }

This code is used in section 1184.

1189.   ⟨Compile a ligature/kern command 1189⟩ ≡
  {
    next_char(*mp⃗nl*) = *qi*(*c*);
    skip_byte(*mp⃗nl*) = *qi*(0);
    if (*cur_mod*( ) < 128) {   /∗ ligature op ∗/
       op_byte(*mp⃗nl*) = *qi*(*cur_mod*( ));
       rem_byte(*mp⃗nl*) = *qi*(*mp_get_code*(*mp*));
    }
    else {
       mp_get_x_next(*mp*);
       mp_scan_expression(*mp*);
       if (*mp⃗cur_exp.type* ≠ *mp_known*) {
          const char ∗*hlp*[ ] = {"The␣amount␣of␣kern␣should␣be␣a␣known␣numeric␣value.",
             "I'm␣zeroing␣this␣one.␣Proceed,␣with␣fingers␣crossed.", Λ};
          mp_disp_err(*mp*, Λ);
          set_number_to_zero(*new_expr.data.n*);
          mp_back_error(*mp*, "Improper␣kern", *hlp*, *true*);
          ;
          mp_get_x_next(*mp*);
          mp_flush_cur_exp(*mp*, *new_expr*);
       }
       number_clone(*mp⃗kern*[*mp⃗nk*], *cur_exp_value_number*( ));
       k = 0;
       while (¬*number_equal*(*mp⃗kern*[*k*], *cur_exp_value_number*( ))) *incr*(*k*);
       if (*k* ≡ *mp⃗nk*) {
          if (*mp⃗nk* ≡ *max_tfm_int*) *mp_fatal_error*(*mp*, "too␣many␣TFM␣kerns");
          mp⃗nk ++;
       }
       op_byte(*mp⃗nl*) = *qi*(*kern_flag* + (*k*/256));
       rem_byte(*mp⃗nl*) = *qi*((*k* % 256));
    }
    mp⃗lk_started = *true*;
  }
This code is used in section 1184.

1190.   **#define**   *missing_extensible_punctuation*(*A*)
            {
               char *msg*[256];
               const char *∗hlp*[ ] = {"I'm␣processing␣'extensible␣c:␣t,m,b,r'.", Λ};

               mp_snprintf(*msg*, 256, "Missing␣%s␣has␣been␣inserted", (*A*));
               mp_back_error(*mp*, *msg*, *hlp*, *true*);
            }

⟨ Define an extensible recipe 1190 ⟩ ≡
  {
    if (*mp*→*ne* ≡ 256) *mp_fatal_error*(*mp*, "too␣many␣extensible␣recipies");
    c = *mp_get_code*(*mp*);
    mp_set_tag(*mp*, *c*, *ext_tag*, *mp*→*ne*);
    if (*cur_cmd*( ) ≠ *mp_colon*) *missing_extensible_punctuation*(":");
    ext_top(*mp*→*ne*) = *qi*(*mp_get_code*(*mp*));
    if (*cur_cmd*( ) ≠ *mp_comma*) *missing_extensible_punctuation*(",");
    ext_mid(*mp*→*ne*) = *qi*(*mp_get_code*(*mp*));
    if (*cur_cmd*( ) ≠ *mp_comma*) *missing_extensible_punctuation*(",");
    ext_bot(*mp*→*ne*) = *qi*(*mp_get_code*(*mp*));
    if (*cur_cmd*( ) ≠ *mp_comma*) *missing_extensible_punctuation*(",");
    ext_rep(*mp*→*ne*) = *qi*(*mp_get_code*(*mp*));
    mp→*ne* ++;
  }

This code is used in section 1183.

1191.   The header could contain ASCII zeroes, so can't use *strdup*.

⟨ Store a list of header bytes 1191 ⟩ ≡
  j−−; **do**
  {
    if (*j* ≥ *mp*→*header_size*) {
       size_t *l* = (**size_t**)(*mp*→*header_size* + (*mp*→*header_size*/4));
       char *∗t* = *xmalloc*(*l*, 1);

       memset(*t*, 0, *l*);
       (**void**) *memcpy*(*t*, *mp*→*header_byte*, (**size_t**) *mp*→*header_size*);
       xfree(*mp*→*header_byte*);
       mp→*header_byte* = *t*;
       mp→*header_size* = (**int**) *l*;
    }
    mp→*header_byte*[*j*] = (**char**) *mp_get_code*(*mp*);
    incr(*j*);
    incr(*mp*→*header_last*);
  }
  while (*cur_cmd*( ) ≡ *mp_comma*)
This code is used in section 1183.

1192.   ⟨Store a list of font dimensions 1192⟩ ≡
  do
  {
    if $(j > \mathit{max\_tfm\_int})$ $\mathit{mp\_fatal\_error}(mp, \texttt{"too}_{\sqcup}\texttt{many}_{\sqcup}\texttt{fontdimens"})$;
    while $(j > mp{\rightarrow}np)$ {
      $mp{\rightarrow}np\!+\!+$;
      $\mathit{set\_number\_to\_zero}(mp{\rightarrow}param[mp{\rightarrow}np])$;
    }
    ;
    $\mathit{mp\_get\_x\_next}(mp)$;
    $\mathit{mp\_scan\_expression}(mp)$;
    if $(mp{\rightarrow}cur\_exp.type \neq mp\_known)$ {
      const char $*hlp[\,] = \{\texttt{"I'm}_{\sqcup}\texttt{zeroing}_{\sqcup}\texttt{this}_{\sqcup}\texttt{one.}_{\sqcup}\texttt{Proceed,}_{\sqcup}\texttt{with}_{\sqcup}\texttt{fingers}_{\sqcup}\texttt{crossed."}, \Lambda\}$;
      $\mathit{mp\_disp\_err}(mp, \Lambda)$;
      $\mathit{set\_number\_to\_zero}(new\_expr.data.n)$;
      $\mathit{mp\_back\_error}(mp, \texttt{"Improper}_{\sqcup}\texttt{font}_{\sqcup}\texttt{parameter"}, hlp, true)$;
      ;
      $\mathit{mp\_get\_x\_next}(mp)$;
      $\mathit{mp\_flush\_cur\_exp}(mp, new\_expr)$;
    }
    $\mathit{number\_clone}(mp{\rightarrow}param[j], cur\_exp\_value\_number(\,))$;
    $\mathit{incr}(j)$;
  }
  while $(cur\_cmd(\,) \equiv mp\_comma)$

This code is used in section 1183.

1193.   OK: We've stored all the data that is needed for the TFM file. All that remains is to output it in the correct format.

An interesting problem needs to be solved in this connection, because the TFM format allows at most 256 widths, 16 heights, 16 depths, and 64 italic corrections. If the data has more distinct values than this, we want to meet the necessary restrictions by perturbing the given values as little as possible.

METAPOST solves this problem in two steps. First the values of a given kind (widths, heights, depths, or italic corrections) are sorted; then the list of sorted values is perturbed, if necessary.

The sorting operation is facilitated by having a special node of essentially infinite *value* at the end of the current list.

⟨Initialize table entries 182⟩ +≡
  $mp{\rightarrow}inf\_val = mp\_get\_value\_node(mp)$;
  $\mathit{set\_value\_number}(mp{\rightarrow}inf\_val, fraction\_four\_t)$;

1194.   ⟨Free table entries 183⟩ +≡
  $\mathit{mp\_free\_value\_node}(mp, mp{\rightarrow}inf\_val)$;

1195.   Straight linear insertion is good enough for sorting, since the lists are usually not terribly long. As we work on the data, the current list will start at *mp_link*(*temp_head*) and end at *inf_val*; the nodes in this list will be in increasing order of their *value* fields.

Given such a list, the *sort_in* function takes a value and returns a pointer to where that value can be found in the list. The value is inserted in the proper place, if necessary.

At the time we need to do these operations, most of METAPOST's work has been completed, so we will have plenty of memory to play with. The value nodes that are allocated for sorting will never be returned to free storage.

#define *clear_the_list*   *mp_link*(*mp*→*temp_head*) = *mp*→*inf_val*

```
static mp_node mp_sort_in(MP mp, mp_number v)
{
  mp_node p, q, r;       /* list manipulation registers */
  p = mp→temp_head;
  while (1) {
    q = mp_link(p);
    if (number_lessequal(v, value_number(q))) break;
    p = q;
  }
  if (number_less(v, value_number(q))) {
    r = mp_get_value_node(mp);
    set_value_number(r, v);
    mp_link(r) = q;
    mp_link(p) = r;
  }
  return mp_link(p);
}
```

1196.   Now we come to the interesting part, where we reduce the list if necessary until it has the required size. The *min_cover* routine is basic to this process; it computes the minimum number m such that the values of the current sorted list can be covered by m intervals of width d. It also sets the global value *perturbation* to the smallest value $d' > d$ such that the covering found by this algorithm would be different.

In particular, *min_cover*(0) returns the number of distinct values in the current list and sets *perturbation* to the minimum distance between adjacent values.

```
static integer mp_min_cover(MP mp, mp_number d)
{
    mp_node p;        /* runs through the current list */
    mp_number l;        /* the least element covered by the current interval */
    mp_number test;
    integer m;        /* lower bound on the size of the minimum cover */
    m = 0;
    new_number(l);
    new_number(test);
    p = mp_link(mp→temp_head);
    set_number_to_inf(mp→perturbation);
    while (p ≠ mp→inf_val) {
        incr(m);
        number_clone(l, value_number(p));
        do {
            p = mp_link(p);
            set_number_from_addition(test, l, d);
        } while (number_lessequal(value_number(p), test));
        set_number_from_substraction(test, value_number(p), l);
        if (number_less(test, mp→perturbation)) {
            number_clone(mp→perturbation, value_number(p));
            number_substract(mp→perturbation, l);
        }
    }
    free_number(test);
    free_number(l);
    return m;
}
```

1197.   ⟨ Global variables 14 ⟩ +≡
  mp_number *perturbation*;   /* quantity related to TFM rounding */
  integer *excess*;   /* the list is this much too long */

1198.   ⟨ Initialize table entries 182 ⟩ +≡
  new_number(*mp→perturbation*);

1199.   ⟨ Dealloc variables 27 ⟩ +≡
  free_number(*mp→perturbation*);

1200.   The smallest d such that a given list can be covered with m intervals is determined by the *threshold* routine, which is sort of an inverse to *min_cover*. The idea is to increase the interval size rapidly until finding the range, then to go sequentially until the exact borderline has been discovered.

```
static void mp_threshold (MP mp, mp_number ret, integer m)
{
  mp_number d, arg1 ;      /∗ lower bound on the smallest interval size ∗/
  new_number (d);
  new_number (arg1 );
  mp→excess = mp_min_cover (mp, zero_t) − m;
  if (mp→excess ≤ 0) {
    number_clone (ret, zero_t);
  }
  else {
    do {
      number_clone (d, mp→perturbation );
      set_number_from_addition (arg1 , d, d);
    } while (mp_min_cover (mp, arg1 ) > m);
    while (mp_min_cover (mp, d) > m) {
      number_clone (d, mp→perturbation );
    }
    number_clone (ret, d);
  }
  free_number (d);
  free_number (arg1 );
}
```

1201.   The *skimp* procedure reduces the current list to at most m entries, by changing values if necessary. It also sets *indep_value*(p): = k if *value*(p) is the kth distinct value on the resulting list, and it sets *perturbation* to the maximum amount by which a *value* field has been changed. The size of the resulting list is returned as the value of *skimp*.

```
static integer mp_skimp(MP mp, integer m)
{
    mp_number d;        /* the size of intervals being coalesced */
    mp_node p, q, r;      /* list manipulation registers */
    mp_number l;        /* the least value in the current interval */
    mp_number v;        /* a compromise value */
    mp_number l_d;

    new_number(d);
    mp_threshold(mp, d, m);
    new_number(l);
    new_number(l_d);
    new_number(v);
    set_number_to_zero(mp→perturbation);
    q = mp→temp_head;
    m = 0;
    p = mp_link(mp→temp_head);
    while (p ≠ mp→inf_val) {
        incr(m);
        number_clone(l, value_number(p));
        set_indep_value(p, m);
        set_number_from_addition(l_d, l, d);
        if (number_lessequal(value_number(mp_link(p)), l_d)) {
            ⟨Replace an interval of values by its midpoint 1202⟩;
        }
        q = p;
        p = mp_link(p);
    }
    free_number(l_d);
    free_number(d);
    free_number(l);
    free_number(v);
    return m;
}
```

1202.   ⟨Replace an interval of values by its midpoint 1202⟩ ≡
```
{
    mp_number test;
    new_number(test);
    do {
        p = mp_link(p);
        set_indep_value(p, m);
        decr(mp→excess);
        if (mp→excess ≡ 0) {
            number_clone(l_d, l);
        }
    } while (number_lessequal(value_number(mp_link(p)), l_d));
    set_number_from_substraction(test, value_number(p), l);
    number_halfp(test);
    set_number_from_addition(v, l, test);
    set_number_from_substraction(test, value_number(p), v);
    if (number_greater(test, mp→perturbation)) number_clone(mp→perturbation, test);
    r = q;
    do {
        r = mp_link(r);
        set_value_number(r, v);
    } while (r ≠ p);
    mp_link(q) = p;     /∗ remove duplicate values from the current list ∗/
    free_number(test);
}
```
This code is used in section 1201.

1203.   A warning message is issued whenever something is perturbed by more than 1/16 pt.
```
static void mp_tfm_warning(MP mp, quarterword m)
{
    mp_print_nl(mp, "(some␣");
    mp_print(mp, internal_name(m));
    ;
    mp_print(mp, "␣values␣had␣to␣be␣adjusted␣by␣as␣much␣as␣");
    print_number(mp→perturbation);
    mp_print(mp, "pt)");
}
```

1204.   Here's an example of how we use these routines. The width data needs to be perturbed only if there are 256 distinct widths, but METAPOST must check for this case even though it is highly unusual.

An integer variable k will be defined when we use this code. The *dimen_head* array will contain pointers to the sorted lists of dimensions.

#define *tfm_warn_threshold_k*   ((**math_data** ∗) *mp→math*)*→tfm_warn_threshold_t*

⟨ Massage the TFM widths 1204 ⟩ ≡
  clear_the_list;
  for ($k = mp\text{→}bc$; $k \le mp\text{→}ec$; $k{+}{+}$) {
    if (*mp→char_exists*[k]) *mp→tfm_width*[k] = *mp_sort_in*(*mp*, *value_number*(*mp→tfm_width*[k]));
  }
  mp→nw = (**short**)(*mp_skimp*(*mp*, 255) + 1);
  mp→dimen_head[1] = *mp_link*(*mp→temp_head*); **if** (*number_greaterequal*(*mp→perturbation*,
    tfm_warn_threshold_k)) *mp_tfm_warning*(*mp*, *mp_char_wd*)
This code is used in section 1291.

1205.   ⟨ Global variables 14 ⟩ +≡
  mp_node *dimen_head*[5];   /∗ lists of TFM dimensions ∗/

1206.   Heights, depths, and italic corrections are different from widths not only because their list length
is more severely restricted, but also because zero values do not need to be put into the lists.

⟨ Massage the TFM heights, depths, and italic corrections 1206 ⟩ ≡
      clear_the_list;
      for (*k* = *mp*→*bc*; *k* ≤ *mp*→*ec*; *k*++) {
            if (*mp*→*char_exists*[*k*]) {
                  if (*mp*→*tfm_height*[*k*] ≡ 0) *mp*→*tfm_height*[*k*] = *mp*→*zero_val*;
                  else *mp*→*tfm_height*[*k*] = *mp_sort_in*(*mp*, *value_number*(*mp*→*tfm_height*[*k*]));
            }
      }
      mp→*nh* = (**short**)(*mp_skimp*(*mp*, 15) + 1);
      mp→*dimen_head*[2] = *mp_link*(*mp*→*temp_head*);
      if (*number_greaterequal*(*mp*→*perturbation*, *tfm_warn_threshold_k*)) *mp_tfm_warning*(*mp*, *mp_char_ht*);
      clear_the_list;
      for (*k* = *mp*→*bc*; *k* ≤ *mp*→*ec*; *k*++) {
            if (*mp*→*char_exists*[*k*]) {
                  if (*mp*→*tfm_depth*[*k*] ≡ 0) *mp*→*tfm_depth*[*k*] = *mp*→*zero_val*;
                  else *mp*→*tfm_depth*[*k*] = *mp_sort_in*(*mp*, *value_number*(*mp*→*tfm_depth*[*k*]));
            }
      }
      mp→*nd* = (**short**)(*mp_skimp*(*mp*, 15) + 1);
      mp→*dimen_head*[3] = *mp_link*(*mp*→*temp_head*);
      if (*number_greaterequal*(*mp*→*perturbation*, *tfm_warn_threshold_k*)) *mp_tfm_warning*(*mp*, *mp_char_dp*);
      clear_the_list;
      for (*k* = *mp*→*bc*; *k* ≤ *mp*→*ec*; *k*++) {
            if (*mp*→*char_exists*[*k*]) {
                  if (*mp*→*tfm_ital_corr*[*k*] ≡ 0) *mp*→*tfm_ital_corr*[*k*] = *mp*→*zero_val*;
                  else *mp*→*tfm_ital_corr*[*k*] = *mp_sort_in*(*mp*, *value_number*(*mp*→*tfm_ital_corr*[*k*]));
            }
      }
      mp→*ni* = (**short**)(*mp_skimp*(*mp*, 63) + 1);
      mp→*dimen_head*[4] = *mp_link*(*mp*→*temp_head*); **if** (*number_greaterequal*(*mp*→*perturbation*,
            tfm_warn_threshold_k)) *mp_tfm_warning*(*mp*, *mp_char_ic*)
This code is used in section 1291.

1207.   ⟨ Initialize table entries 182 ⟩ +≡
      mp→*zero_val* = *mp_get_value_node*(*mp*);
      set_value_number(*mp*→*zero_val*, *zero_t*);

1208.   ⟨ Free table entries 183 ⟩ +≡
      mp_free_value_node(*mp*, *mp*→*zero_val*);

1209.   Bytes 5–8 of the header are set to the design size, unless the user has some crazy reason for specifying them differently.

Error messages are not allowed at the time this procedure is called, so a warning is printed instead.

The value of *max_tfm_dimen* is calculated so that

$$make\_scaled(16 * max\_tfm\_dimen, internal\_value(mp\_design\_size)) < three\_bytes.$$

#define *three_bytes* °*100000000*   /* 2^{24} */

```
static void mp_fix_design_size(MP mp)
{
    mp_number d;      /* the design size */

    new_number(d);
    number_clone(d, internal_value(mp_design_size));
    if (number_less(d, unity_t) ∨ number_greaterequal(d, fraction_half_t)) {
        if (¬number_zero(d)) mp_print_nl(mp, "(illegal␣design␣size␣has␣been␣changed␣to␣128pt)");
        ;
        set_number_from_scaled(d, °40000000);
        number_clone(internal_value(mp_design_size), d);
    }
    if (mp→header_byte[4] ≡ 0 ∧ mp→header_byte[5] ≡ 0 ∧ mp→header_byte[6] ≡ 0 ∧ mp→header_byte[7] ≡ 0) {
        integer dd = number_to_scaled(d);

        mp→header_byte[4] = (char)(dd/°4000000);
        mp→header_byte[5] = (char)((dd/4096) % 256);
        mp→header_byte[6] = (char)((dd/16) % 256);
        mp→header_byte[7] = (char)((dd % 16) * 16);
    }    /* mp→max_tfm_dimen = 16 * internal_value(mp_design_size) − 1 −
            internal_value(mp_design_size)/°10000000 */
    {
        mp_number secondpart;

        new_number(secondpart);
        number_clone(secondpart, internal_value(mp_design_size));
        number_clone(mp→max_tfm_dimen, secondpart);
        number_divide_int(secondpart, °10000000);
        number_multiply_int(mp→max_tfm_dimen, 16);
        number_add_scaled(mp→max_tfm_dimen, −1);
        number_substract(mp→max_tfm_dimen, secondpart);
        free_number(secondpart);
    }
    if (number_greaterequal(mp→max_tfm_dimen, fraction_half_t)) {
        number_clone(mp→max_tfm_dimen, fraction_half_t);
        number_add_scaled(mp→max_tfm_dimen, −1);
    }
    free_number(d);
}
```

1210.   The *dimen_out* procedure computes a *fix_word* relative to the design size. If the data was out of range, it is corrected and the global variable *tfm_changed* is increased by one.

```
static integer mp_dimen_out(MP mp, mp_number x_orig)
{
  integer ret;
  mp_number abs_x;
  mp_number x;
  new_number(abs_x);
  new_number(x);
  number_clone(x, x_orig);
  number_clone(abs_x, x_orig);
  number_abs(abs_x);
  if (number_greater(abs_x, mp→max_tfm_dimen)) {
    incr(mp→tfm_changed);
    if (number_positive(x))  number_clone(x, mp→max_tfm_dimen);
    else {
      number_clone(x, mp→max_tfm_dimen);
      number_negate(x);
    }
  }
  {
    mp_number arg1;
    new_number(arg1);
    number_clone(arg1, x);
    number_multiply_int(arg1, 16);
    make_scaled(x, arg1, internal_value(mp_design_size));
    free_number(arg1);
  }
  free_number(abs_x);
  ret = number_to_scaled(x);
  free_number(x);
  return ret;
}
```

1211.   ⟨Global variables 14⟩ +≡
mp_number *max_tfm_dimen*;   /∗ bound on widths, heights, kerns, etc. ∗/
integer *tfm_changed*;   /∗ the number of data entries that were out of bounds ∗/

1212.   ⟨Initialize table entries 182⟩ +≡
new_number(mp→max_tfm_dimen);

1213.   ⟨Dealloc variables 27⟩ +≡
free_number(mp→max_tfm_dimen);

1214.   If the user has not specified any of the first four header bytes, the *fix_check_sum* procedure replaces them by a "check sum" computed from the *tfm_width* data relative to the design size.

> **static void** *mp_fix_check_sum*(**MP** *mp*)
> {
>   **eight_bits** *k*;   /∗ runs through character codes ∗/
>   **eight_bits** B1, B2, B3, B4;   /∗ bytes of the check sum ∗/
>   **integer** *x*;   /∗ hash value used in check sum computation ∗/
>
>   **if** (*mp*→*header_byte*[0] ≡ 0 ∧ *mp*→*header_byte*[1] ≡ 0 ∧ *mp*→*header_byte*[2] ≡ 0 ∧ *mp*→*header_byte*[3] ≡ 0) {
>   ⟨ Compute a check sum in (*b1*, *b2*, *b3*, *b4*) 1215 ⟩;
>   *mp*→*header_byte*[0] = (**char**) B1;
>   *mp*→*header_byte*[1] = (**char**) B2;
>   *mp*→*header_byte*[2] = (**char**) B3;
>   *mp*→*header_byte*[3] = (**char**) B4;
>   **return**;
>   }
> }

1215.   ⟨ Compute a check sum in (*b1*, *b2*, *b3*, *b4*) 1215 ⟩ ≡
> B1 = *mp*→*bc*;
> B2 = *mp*→*ec*;
> B3 = *mp*→*bc*;
> B4 = *mp*→*ec*;
> *mp*→*tfm_changed* = 0;
> **for** (*k* = *mp*→*bc*; *k* ≤ *mp*→*ec*; *k*++) {
>   **if** (*mp*→*char_exists*[*k*]) {
>   *x* = *mp_dimen_out*(*mp*, *value_number*(*mp*→*tfm_width*[*k*])) + (*k* + 4) ∗ °*20000000*;
>   /∗ this is positive ∗/
>   B1 = (**eight_bits**)((B1 + B1 + *x*) % 255);
>   B2 = (**eight_bits**)((B2 + B2 + *x*) % 253);
>   B3 = (**eight_bits**)((B3 + B3 + *x*) % 251);
>   B4 = (**eight_bits**)((B4 + B4 + *x*) % 247);
>   }
>   **if** (*k* ≡ *mp*→*ec*) **break**;
> }

This code is used in section 1214.

1216.   Finally we're ready to actually write the TFM information. Here are some utility routines for this purpose.

#define $tfm\_out(A)$ **do**
        {   /∗ output one byte to tfm_file ∗/
            unsigned char $s = ($**unsigned char**$)(A);$

            $(mp \text{→} write\_binary\_file)(mp, mp \text{→} tfm\_file, ($**void** ∗$) \& s, 1);$
        }
        while (0)

static void $mp\_tfm\_two($**MP** $mp,$ **integer** $x)$
{   /∗ output two bytes to tfm_file ∗/
  $tfm\_out(x/256);$
  $tfm\_out(x \% 256);$
}

static void $mp\_tfm\_four($**MP** $mp,$ **integer** $x)$
{   /∗ output four bytes to tfm_file ∗/
  if $(x \geq 0)$ $tfm\_out(x/three\_bytes);$
  else {
    $x = x + °10000000000;$ /∗ use two's complement for negative values ∗/
    $x = x + °10000000000;$
    $tfm\_out((x/three\_bytes) + 128);$
  }
  ;
  $x = x \% three\_bytes;$
  $tfm\_out(x/number\_to\_scaled(unity\_t));$
  $x = x \% number\_to\_scaled(unity\_t);$
  $tfm\_out(x/°400);$
  $tfm\_out(x \% °400);$
}

static void $mp\_tfm\_qqqq($**MP** $mp,$ **four_quarters** $x)$
{   /∗ output four quarterwords to tfm_file ∗/
  $tfm\_out(qo(x.b0));$
  $tfm\_out(qo(x.b1));$
  $tfm\_out(qo(x.b2));$
  $tfm\_out(qo(x.b3));$
}

1217.   ⟨Finish the TFM file 1217⟩ ≡

  if $(mp\text{-}job\_name \equiv \Lambda)$ $mp\_open\_log\_file(mp)$;

  $mp\_pack\_job\_name(mp, \texttt{".tfm"})$;

  while $(\neg mp\_open\_out(mp, \&mp\text{-}tfm\_file, mp\_filetype\_metrics))$

    $mp\_prompt\_file\_name(mp, \texttt{"file␣name␣for␣font␣metrics"}, \texttt{".tfm"})$;

  $mp\text{-}metric\_file\_name = xstrdup(mp\text{-}name\_of\_file)$;

  ⟨Output the subfile sizes and header bytes 1218⟩;

  ⟨Output the character information bytes, then output the dimensions themselves 1219⟩;

  ⟨Output the ligature/kern program 1222⟩;

  ⟨Output the extensible character recipes and the font metric parameters 1223⟩;

  if $(number\_positive(internal\_value(mp\_tracing\_stats)))$ ⟨Log the subfile sizes of the TFM file 1224⟩;

  $mp\_print\_nl(mp, \texttt{"Font␣metrics␣written␣on␣"})$;

  $mp\_print(mp, mp\text{-}metric\_file\_name)$;

  $mp\_print\_char(mp, xord(\texttt{'.'}))$;

  ; $(mp\text{-}close\_file)(mp, mp\text{-}tfm\_file)$

This code is used in section 1291.

1218.   Integer variables lh, k, and lk_offset will be defined when we use this code.

⟨Output the subfile sizes and header bytes 1218⟩ ≡

  $k = mp\text{-}header\_last$;

  $\texttt{LH} = (k+4)/4$;   /∗ this is the number of header words ∗/

  if $(mp\text{-}bc > mp\text{-}ec)$ $mp\text{-}bc = 1$;   /∗ if there are no characters, $ec = 0$ and $bc = 1$ ∗/

  ⟨Compute the ligature/kern program offset and implant the left boundary label 1220⟩;

  $mp\_tfm\_two(mp, 6 + \texttt{LH} + (mp\text{-}ec - mp\text{-}bc + 1) + mp\text{-}nw + mp\text{-}nh + mp\text{-}nd + mp\text{-}ni + mp\text{-}nl + lk\_offset +$

    $mp\text{-}nk + mp\text{-}ne + mp\text{-}np)$;   /∗ this is the total number of file words that will be output ∗/

  $mp\_tfm\_two(mp, \texttt{LH})$;

  $mp\_tfm\_two(mp, mp\text{-}bc)$;

  $mp\_tfm\_two(mp, mp\text{-}ec)$;

  $mp\_tfm\_two(mp, mp\text{-}nw)$;

  $mp\_tfm\_two(mp, mp\text{-}nh)$;

  $mp\_tfm\_two(mp, mp\text{-}nd)$;

  $mp\_tfm\_two(mp, mp\text{-}ni)$;

  $mp\_tfm\_two(mp, mp\text{-}nl + lk\_offset)$;

  $mp\_tfm\_two(mp, mp\text{-}nk)$;

  $mp\_tfm\_two(mp, mp\text{-}ne)$;

  $mp\_tfm\_two(mp, mp\text{-}np)$;

  for $(k = 0;\ k < 4 * \texttt{LH};\ k{+}{+})$ {

    $tfm\_out(mp\text{-}header\_byte[k])$;

  }

This code is used in section 1217.

1219. ⟨Output the character information bytes, then output the dimensions themselves 1219⟩ ≡
```
    for (k = mp→bc; k ≤ mp→ec; k++) {
      if (¬mp→char_exists[k]) {
        mp_tfm_four(mp, 0);
      }
      else {
        tfm_out(indep_value(mp→tfm_width[k]));      /∗ the width index ∗/
        tfm_out((indep_value(mp→tfm_height[k])) ∗ 16 + indep_value(mp→tfm_depth[k]));
        tfm_out((indep_value(mp→tfm_ital_corr[k])) ∗ 4 + mp→char_tag[k]);
        tfm_out(mp→char_remainder[k]);
      }
      ;
    }
    mp→tfm_changed = 0;
    for (k = 1; k ≤ 4; k++) {
      mp_tfm_four(mp, 0);
      p = mp→dimen_head[k];
      while (p ≠ mp→inf_val) {
        mp_tfm_four(mp, mp_dimen_out(mp, value_number(p)));
        p = mp_link(p);
      }
    }
```
This code is used in section 1217.

1220. We need to output special instructions at the beginning of the *lig_kern* array in order to specify the right boundary character and/or to handle starting addresses that exceed 255. The *label_loc* and *label_char* arrays have been set up to record all the starting addresses; we have $-1 = label\_loc[0] < label\_loc[1] \le \cdots \le label\_loc[label\_ptr]$.

⟨Compute the ligature/kern program offset and implant the left boundary label 1220⟩ ≡
```
    mp→bchar = round_unscaled(internal_value(mp_boundary_char));
    if ((mp→bchar < 0) ∨ (mp→bchar > 255)) {
      mp→bchar = −1;
      mp→lk_started = false;
      lk_offset = 0;
    }
    else {
      mp→lk_started = true;
      lk_offset = 1;
    }
```
⟨Find the minimum *lk_offset* and adjust all remainders 1221⟩;
```
    if (mp→bch_label < undefined_label) {
      skip_byte(mp→nl) = qi(255);
      next_char(mp→nl) = qi(0);
      op_byte(mp→nl) = qi(((mp→bch_label + lk_offset)/256));
      rem_byte(mp→nl) = qi(((mp→bch_label + lk_offset) % 256));
      mp→nl++;      /∗ possibly nl = lig_table_size + 1 ∗/
    }
```
This code is used in section 1218.

1221. ⟨ Find the minimum lk_offset and adjust all remainders 1221 ⟩ ≡
  $k = mp{\rightarrow}label\_ptr$;   /∗ pointer to the largest unallocated label ∗/
  if ($mp{\rightarrow}label\_loc[k] + lk\_offset > 255$) {
    $lk\_offset = 0$;
    $mp{\rightarrow}lk\_started = false$;   /∗ location 0 can do double duty ∗/
    do {
      $mp{\rightarrow}char\_remainder[mp{\rightarrow}label\_char[k]] = lk\_offset$;
      while ($mp{\rightarrow}label\_loc[k-1] \equiv mp{\rightarrow}label\_loc[k]$) {
        $decr(k)$;
        $mp{\rightarrow}char\_remainder[mp{\rightarrow}label\_char[k]] = lk\_offset$;
      }
      $incr(lk\_offset)$;
      $decr(k)$;
    } **while** ($\neg(lk\_offset + mp{\rightarrow}label\_loc[k] < 256)$);   /∗ N.B.: $lk\_offset = 256$ satisfies this when $k = 0$ ∗/
  }
  if ($lk\_offset > 0$) {
    while ($k > 0$) {
      $mp{\rightarrow}char\_remainder[mp{\rightarrow}label\_char[k]] = mp{\rightarrow}char\_remainder[mp{\rightarrow}label\_char[k]] + lk\_offset$;
      $decr(k)$;
    }
  }

This code is used in section 1220.

1222.   ⟨Output the ligature/kern program 1222⟩ ≡
```
  for (k = 0;  k ≤ 255;  k++) {
    if (mp→skip_table[k] < undefined_label) {
       mp_print_nl(mp, "(local␣label␣");
       mp_print_int(mp, k);
       mp_print(mp, "::␣was␣missing)");
       ;
       cancel_skips(mp→skip_table[k]);
    }
  }
  if (mp→lk_started) {        /* lk_offset = 1 for the special bchar */
    tfm_out(255);
    tfm_out(mp→bchar);
    mp_tfm_two(mp, 0);
  }
  else {
    for (k = 1;  k ≤ lk_offset;  k++) {       /* output the redirection specs */
       mp→ll = mp→label_loc[mp→label_ptr];
       if (mp→bchar < 0) {
         tfm_out(254);
         tfm_out(0);
       }
       else {
         tfm_out(255);
         tfm_out(mp→bchar);
       }
       ;
       mp_tfm_two(mp, mp→ll + lk_offset);
       do {
         mp→label_ptr −−;
       } while (¬(mp→label_loc[mp→label_ptr] < mp→ll));
    }
  }
  for (k = 0;  k < mp→nl;  k++)  mp_tfm_qqqq(mp, mp→lig_kern[k]);
  {
    mp_number arg;

    new_number(arg);
    for (k = 0;  k < mp→nk;  k++) {
       number_clone(arg, mp→kern[k]);
       mp_tfm_four(mp, mp_dimen_out(mp, arg));
    }
    free_number(arg);
  }
```
This code is used in section 1217.

1223.   ⟨Output the extensible character recipes and the font metric parameters 1223⟩ ≡
  for $(k = 0; \ k < mp\text{-}ne; \ k\text{++})$ $mp\_tfm\_qqqq(mp, mp\text{-}exten[k])$;
  {
    mp_number arg;

    $new\_number(arg)$;
    for $(k = 1; \ k \le mp\text{-}np; \ k\text{++})$ {
      if $(k \equiv 1)$ {
        $number\_clone(arg, mp\text{-}param[1])$;
        $number\_abs(arg)$;
        if $(number\_less(arg, fraction\_half\_t))$ {
          $mp\_tfm\_four(mp, number\_to\_scaled(mp\text{-}param[1]) * 16)$;
        }
        else {
          $incr(mp\text{-}tfm\_changed)$;
          if $(number\_positive(mp\text{-}param[1]))$ $mp\_tfm\_four(mp, max\_integer)$;
          else $mp\_tfm\_four(mp, -max\_integer)$;
        }
      }
      else {
        $number\_clone(arg, mp\text{-}param[k])$;
        $mp\_tfm\_four(mp, mp\_dimen\_out(mp, arg))$;
      }
    }
    $free\_number(arg)$;
  }
  if $(mp\text{-}tfm\_changed > 0)$ {
    if $(mp\text{-}tfm\_changed \equiv 1)$ {
      $mp\_print\_nl(mp, \texttt{"(a\textvisiblespace font\textvisiblespace metric\textvisiblespace dimension"})$;
    }
    else {
      $mp\_print\_nl(mp, \texttt{"("})$;
      $mp\_print\_int(mp, mp\text{-}tfm\_changed)$;
      ;
      $mp\_print(mp, \texttt{"\textvisiblespace font\textvisiblespace metric\textvisiblespace dimensions"})$;
    }
    $mp\_print(mp, \texttt{"\textvisiblespace had\textvisiblespace to\textvisiblespace be\textvisiblespace decreased)"})$;
  }
This code is used in section 1217.

1224.   ⟨Log the subfile sizes of the TFM file 1224⟩ ≡
  {
    char $s[200]$;

    $wlog\_ln(\texttt{"\textvisiblespace"})$;
    if $(mp\text{-}bch\_label < undefined\_label)$ $mp\text{-}nl\text{--}$;
    $mp\_snprintf(s, 128, \texttt{"(You\textvisiblespace used\textvisiblespace\%iw,\%ih,\%id,\%ii,\%il,\%ik,\%ie,\%ip\textvisiblespace metric\textvisiblespace file\textvisiblespace positions)"}$,
        $mp\text{-}nw, mp\text{-}nh, mp\text{-}nd, mp\text{-}ni, mp\text{-}nl, mp\text{-}nk, mp\text{-}ne, mp\text{-}np)$;
    $wlog\_ln(s)$;
  }
This code is used in section 1217.

1225.   Reading font metric data.

METAPOST isn't a typesetting program but it does need to find the bounding box of a sequence of typeset characters. Thus it needs to read TFM files as well as write them.

⟨ Global variables 14 ⟩ +≡
   void *∗tfm_infile*;

1226. All the width, height, and depth information is stored in an array called *font_info*. This array is allocated sequentially and each font is stored as a series of *char_info* words followed by the width, height, and depth tables. Since *font_name* entries are permanent, their *str_ref* values are set to MAX_STR_REF.

⟨ Types in the outer block 33 ⟩ +≡
   typedef unsigned int font_number;   /∗ 0..$_\text{F}$ont_max ∗/

1227. The *font_info* array is indexed via a group directory arrays. For example, the *char_info* data for character *c* in font *f* will be in *font_info*[*char_base*[*f*] + *c*].*qqqq*.

⟨ Global variables 14 ⟩ +≡
   font_number *font_max*;   /∗ maximum font number for included text fonts ∗/
   size_t *font_mem_size*;   /∗ number of words for TFM information for text fonts ∗/
   font_data *∗font_info*;   /∗ height, width, and depth data ∗/
   char *∗∗font_enc_name*;   /∗ encoding names, if any ∗/
   boolean *∗font_ps_name_fixed*;   /∗ are the postscript names fixed already? ∗/
   size_t *next_fmem*;   /∗ next unused entry in *font_info* ∗/
   font_number *last_fnum*;   /∗ last font number used so far ∗/
   integer *∗font_dsize*;   /∗ 16 times the "design" size in PostScript points ∗/
   char *∗∗font_name*;   /∗ name as specified in the **infont** command ∗/
   char *∗∗font_ps_name*;   /∗ PostScript name for use when *internal*[*mp_prologues*] > 0 ∗/
   font_number *last_ps_fnum*;   /∗ last valid *font_ps_name* index ∗/
   eight_bits *∗font_bc*;
   eight_bits *∗font_ec*;   /∗ first and last character code ∗/
   int *∗char_base*;   /∗ base address for *char_info* ∗/
   int *∗width_base*;   /∗ index for zeroth character width ∗/
   int *∗height_base*;   /∗ index for zeroth character height ∗/
   int *∗depth_base*;   /∗ index for zeroth character depth ∗/
   mp_node *∗font_sizes*;

1228. ⟨ Allocate or initialize variables 28 ⟩ +≡
   mp⃗font_mem_size = 10000;
   mp⃗font_info = *xmalloc*((*mp⃗font_mem_size* + 1), **sizeof**(**font_data**));
   memset(*mp⃗font_info*, 0, **sizeof**(**font_data**) ∗ (*mp⃗font_mem_size* + 1));
   mp⃗last_fnum = *null_font*;

1229.   ⟨Dealloc variables 27⟩ +≡
```
for (k = 1; k ≤ (int) mp⃗last_fnum; k++) {
    xfree(mp⃗font_enc_name[k]);
    xfree(mp⃗font_name[k]);
    xfree(mp⃗font_ps_name[k]);
}
for (k = 0; k ≤ 255; k++) {       /* These are disabled for now following a bug-report about double free
        errors. TO BE FIXED, bug tracker id 831 */
    /* mp_free_value_node(mp, mp⃗tfm_width[k]);  mp_free_value_node(mp, mp⃗tfm_height[k]);
        mp_free_value_node(mp, mp⃗tfm_depth[k]);  mp_free_value_node(mp, mp⃗tfm_ital_corr[k]); */
}
xfree(mp⃗font_info);
xfree(mp⃗font_enc_name);
xfree(mp⃗font_ps_name_fixed);
xfree(mp⃗font_dsize);
xfree(mp⃗font_name);
xfree(mp⃗font_ps_name);
xfree(mp⃗font_bc);
xfree(mp⃗font_ec);
xfree(mp⃗char_base);
xfree(mp⃗width_base);
xfree(mp⃗height_base);
xfree(mp⃗depth_base);
xfree(mp⃗font_sizes);
```

1230.
```
void mp_reallocate_fonts(MP mp, font_number l)
{
    font_number f;
    XREALLOC(mp⃗font_enc_name, l, char *);
    XREALLOC(mp⃗font_ps_name_fixed, l, boolean);
    XREALLOC(mp⃗font_dsize, l, integer);
    XREALLOC(mp⃗font_name, l, char *);
    XREALLOC(mp⃗font_ps_name, l, char *);
    XREALLOC(mp⃗font_bc, l, eight_bits);
    XREALLOC(mp⃗font_ec, l, eight_bits);
    XREALLOC(mp⃗char_base, l, int);
    XREALLOC(mp⃗width_base, l, int);
    XREALLOC(mp⃗height_base, l, int);
    XREALLOC(mp⃗depth_base, l, int);
    XREALLOC(mp⃗font_sizes, l, mp_node);
    for (f = (mp⃗last_fnum + 1); f ≤ l; f++) {
        mp⃗font_enc_name[f] = Λ;
        mp⃗font_ps_name_fixed[f] = false;
        mp⃗font_name[f] = Λ;
        mp⃗font_ps_name[f] = Λ;
        mp⃗font_sizes[f] = Λ;
    }
    mp⃗font_max = l;
}
```

1231. ⟨Internal library declarations 10⟩ +≡
> **void** *mp_reallocate_fonts*(**MP** *mp*, **font_number** *l*);

1232.   A *null_font* containing no characters is useful for error recovery. Its *font_name* entry starts out empty but is reset each time an erroneous font is found. This helps to cut down on the number of duplicate error messages without wasting a lot of space.

#define *null_font* 0   /∗ the **font_number** for an empty font ∗/

⟨Set initial values of key variables 38⟩ +≡
> *mp*→*font_dsize*[*null_font*] = 0;
> *mp*→*font_bc*[*null_font*] = 1;
> *mp*→*font_ec*[*null_font*] = 0;
> *mp*→*char_base*[*null_font*] = 0;
> *mp*→*width_base*[*null_font*] = 0;
> *mp*→*height_base*[*null_font*] = 0;
> *mp*→*depth_base*[*null_font*] = 0;
> *mp*→*next_fmem* = 0;
> *mp*→*last_fnum* = *null_font*;
> *mp*→*last_ps_fnum* = *null_font*;
> {
>   **static char** *nullfont_name*[ ] = "nullfont";
>   **static char** *nullfont_psname*[ ] = "";
>
>   *mp*→*font_name*[*null_font*] = *nullfont_name*;
>   *mp*→*font_ps_name*[*null_font*] = *nullfont_psname*;
> }
> *mp*→*font_ps_name_fixed*[*null_font*] = *false*;
> *mp*→*font_enc_name*[*null_font*] = Λ;
> *mp*→*font_sizes*[*null_font*] = Λ;

1233.   Each *char_info* word is of type **four_quarters**. The *b0* field contains the *width index*; the *b1* field contains the height index; the *b2* fields contains the depth index, and the *b3* field used only for temporary storage. (It is used to keep track of which characters occur in an edge structure that is being shipped out.) The corresponding words in the width, height, and depth tables are stored as *scaled* values in units of PostScript points.

   With the macros below, the *char_info* word for character *c* in font *f* is *char_mp_info*(*f*, *c*) and the width is

$$char\_width(f, char\_mp\_info(f, c)).sc.$$

#define *char_mp_info*(*A*, *B*)   *mp*→*font_info*[*mp*→*char_base*[(*A*)] + (*B*)].*qqqq*
#define *char_width*(*A*, *B*)   *mp*→*font_info*[*mp*→*width_base*[(*A*)] + (*B*).*b0*].*sc*
#define *char_height*(*A*, *B*)   *mp*→*font_info*[*mp*→*height_base*[(*A*)] + (*B*).*b1*].*sc*
#define *char_depth*(*A*, *B*)   *mp*→*font_info*[*mp*→*depth_base*[(*A*)] + (*B*).*b2*].*sc*
#define *ichar_exists*(*A*) ((*A*).*b0* > 0)

1234.   When we have a font name and we don't know whether it has been loaded yet, we scan the *font_name* array before calling *read_font_info*.

⟨Declarations 8⟩ +≡
> **static font_number** *mp_find_font*(**MP** *mp*, **char** ∗*f*);

1235.   **font_number** *mp_find_font*(**MP** *mp*, **char** *∗f*)
{
    font_number *n*;
    for $(n = 0; \ n \le mp\text{-}last\_fnum; \ n\text{++})$ {
      if $(mp\_xstrcmp(f, mp\text{-}font\_name[n]) \equiv 0)$ {
        return *n*;
      }
    }
    $n = mp\_read\_font\_info(mp, f)$;
    return *n*;
}

1236.   This is an interface function for getting the width of character, as a double in ps units

    double *mp_get_char_dimension*(**MP** *mp*, **char** *∗fname*, **int** *c*, **int** *t*)
{
    unsigned *n*;
    four_quarters *cc*;
    font_number $f = 0$;
    double $w = -1.0$;
    for $(n = 0; \ n \le mp\text{-}last\_fnum; \ n\text{++})$ {
      if $(mp\_xstrcmp(fname, mp\text{-}font\_name[n]) \equiv 0)$ {
        $f = n$;
        break;
      }
    }
    if $(f \equiv 0)$ **return** 0.0;
    $cc = char\_mp\_info(f, c)$;
    if $(\neg ichar\_exists(cc))$ **return** 0.0;
    if $(t \equiv \text{'w'})$ $w = $ (**double**) *char_width*(*f*, *cc*);
    else if $(t \equiv \text{'h'})$ $w = $ (**double**) *char_height*(*f*, *cc*);
    else if $(t \equiv \text{'d'})$ $w = $ (**double**) *char_depth*(*f*, *cc*);
    return $w/655.35 * (72.27/72)$;
}

1237.   ⟨ Exported function headers 18 ⟩ +≡
    double *mp_get_char_dimension*(**MP** *mp*, **char** *∗fname*, **int** *n*, **int** *t*);

1238.   If we discover that the font doesn't have a requested character, we omit it from the bounding box computation and expect the PostScript interpreter to drop it. This routine issues a warning message if the user has asked for it.

⟨ Declarations 8 ⟩ +≡
    static void *mp_lost_warning*(**MP** *mp*, **font_number** *f*, **int** *k*);

1239.   **void** $mp\_lost\_warning(\mathbf{MP}\ mp, \mathbf{font\_number}\ f, \mathbf{int}\ k)$
  {
    if $(number\_positive(internal\_value(mp\_tracing\_lost\_chars)))$ {
      $mp\_begin\_diagnostic(mp)$;
      if $(mp \rightarrow selector \equiv log\_only)\ incr(mp \rightarrow selector)$;
      $mp\_print\_nl(mp, \texttt{"Missing}_\sqcup\texttt{character:}_\sqcup\texttt{There}_\sqcup\texttt{is}_\sqcup\texttt{no}_\sqcup\texttt{"})$;
      ;
      $mp\_print\_int(mp, k)$;
      $mp\_print(mp, \texttt{"}_\sqcup\texttt{in}_\sqcup\texttt{font}_\sqcup\texttt{"})$;
      $mp\_print(mp, mp \rightarrow font\_name[f])$;
      $mp\_print\_char(mp, xord(\texttt{'!'}))$;
      $mp\_end\_diagnostic(mp, false)$;
    }
  }

1240.   The whole purpose of saving the height, width, and depth information is to be able to find the bounding box of an item of text in an edge structure. The *set_text_box* procedure takes a text node and adds this information.

⟨ Declarations 8 ⟩ +≡
  static void $mp\_set\_text\_box(\mathbf{MP}\ mp, \mathbf{mp\_text\_node}\ p)$;

1241.   **void** $mp\_set\_text\_box(\mathbf{MP}\ mp, \mathbf{mp\_text\_node}\ p)$
  {
    font_number f;   /* $mp\_font\_n(p)$ */
    ASCII_code bc, ec;   /* range of valid characters for font f */
    size_t k, kk;   /* current character and character to stop at */
    four_quarters cc;   /* the *char_info* for the current character */
    mp_number h, d;   /* dimensions of the current character */

    $new\_number(h)$;
    $new\_number(d)$;
    $set\_number\_to\_zero(p \rightarrow width)$;
    $set\_number\_to\_neg\_inf(p \rightarrow height)$;
    $set\_number\_to\_neg\_inf(p \rightarrow depth)$;
    $f = (\mathbf{font\_number})\ mp\_font\_n(p)$;
    $bc = mp \rightarrow font\_bc[f]$;
    $ec = mp \rightarrow font\_ec[f]$;
    $kk = mp\_text\_p(p) \rightarrow len$;
    $k = 0$;
    while $(k < kk)$ {
      ⟨ Adjust p's bounding box to contain $str\_pool[k]$; advance k 1242 ⟩;
    }
    ⟨ Set the height and depth to zero if the bounding box is empty 1243 ⟩;
    $free\_number(h)$;
    $free\_number(d)$;
  }

1242.   ⟨Adjust p's bounding box to contain $str\_pool[k]$; advance k 1242⟩ ≡
{
    if $((*(mp\_text\_p(p)\text{→}str + k) < bc) \vee (*(mp\_text\_p(p)\text{→}str + k) > ec))$ {
      $mp\_lost\_warning(mp, f, *(mp\_text\_p(p)\text{→}str + k));$
    }
    else {
      $cc = char\_mp\_info(f, *(mp\_text\_p(p)\text{→}str + k));$
      if $(\neg ichar\_exists(cc))$ {
        $mp\_lost\_warning(mp, f, *(mp\_text\_p(p)\text{→}str + k));$
      }
      else {
        $set\_number\_from\_scaled(p\text{→}width, number\_to\_scaled(p\text{→}width) + char\_width(f, cc));$
        $set\_number\_from\_scaled(h, char\_height(f, cc));$
        $set\_number\_from\_scaled(d, char\_depth(f, cc));$
        if $(number\_greater(h, p\text{→}height))$ $number\_clone(p\text{→}height, h);$
        if $(number\_greater(d, p\text{→}depth))$ $number\_clone(p\text{→}depth, d);$
      }
    }
    $incr(k);$
}

This code is used in section 1241.

1243.   Let's hope modern compilers do comparisons correctly when the difference would overflow.

⟨Set the height and depth to zero if the bounding box is empty 1243⟩ ≡
  if $(number\_to\_scaled(p\text{→}height) < -number\_to\_scaled(p\text{→}depth))$ {
    $set\_number\_to\_zero(p\text{→}height);$
    $set\_number\_to\_zero(p\text{→}depth);$
  }

This code is used in section 1241.

1244.   The new primitives fontmapfile and fontmapline.

⟨Declare action procedures for use by $do\_statement$ 1048⟩ +≡
  static void $mp\_do\_mapfile(\textbf{MP}\ mp);$
  static void $mp\_do\_mapline(\textbf{MP}\ mp);$

1245.   **static void** $mp\_do\_mapfile$(**MP** mp)
{
   $mp\_get\_x\_next$(mp);
   $mp\_scan\_expression$(mp);
   if (mp→$cur\_exp.type \neq mp\_string\_type$) {
     ⟨Complain about improper map operation 1246⟩;
   }
   else {
     mp_map_file(mp, cur_exp_str( ));
   }
}
static void $mp\_do\_mapline$(**MP** mp)
{
   $mp\_get\_x\_next$(mp);
   $mp\_scan\_expression$(mp);
   if (mp→$cur\_exp.type \neq mp\_string\_type$) {
     ⟨Complain about improper map operation 1246⟩;
   }
   else {
     mp_map_line(mp, cur_exp_str( ));
   }
}

1246.   ⟨Complain about improper map operation 1246⟩ ≡
{
   const char $*hlp$[ ] = {"Only␣known␣strings␣can␣be␣map␣files␣or␣map␣lines.", Λ};
   mp_disp_err(mp, Λ);
   mp_back_error(mp, "Unsuitable␣expression", hlp, $true$);
   $mp\_get\_x\_next$(mp);
}
This code is used in section 1245.

1247.   To print *scaled* value to PDF output we need some subroutines to ensure accurary.
#define $max\_integer$ #7FFFFFFF   /∗ $2^{31} - 1$ ∗/
⟨Global variables 14⟩ +≡
   integer ten_pow[10];   /∗ $10^0..10^9$ ∗/
   integer $scaled\_out$;   /∗ amount of *scaled* that was taken out in *divide_scaled* ∗/

1248.   ⟨Set initial values of key variables 38⟩ +≡
  mp→ten_pow[0] = 1;
  for ($i = 1$; $i \leq 9$; i++) {
    mp→ten_pow[i] = $10 * mp$→ten_pow[$i - 1$];
  }

718   SHIPPING PICTURES OUT

MetaPost   §1249

1249.   **Shipping pictures out.**   The *ship_out* procedure, to be described below, is given a pointer to an edge structure. Its mission is to output a file containing the PostScript description of an edge structure.

1250.   Each time an edge structure is shipped out we write a new PostScript output file named according to the current **charcode**.

   This is the only backend function that remains in the main *mpost.w* file. There are just too many variable accesses needed for status reporting etcetera to make it worthwile to move the code to *psout.w*.

⟨Internal library declarations 10⟩ +≡

   void *mp_open_output_file*(**MP** *mp*);
   char *∗mp_get_output_file_name*(**MP** *mp*);
   char *∗mp_set_output_file_name*(**MP** *mp*, **integer** *c*);

1251.   **static void** $mp\_append\_to\_template$(**MP** mp, **integer** ff, **integer** c, **boolean** $rounding$)
{
  if ($internal\_type(c) \equiv mp\_string\_type$) {
    char $*ss = mp\_str(mp, internal\_string(c))$;

    $mp\_print(mp, ss)$;
  }
  else if ($internal\_type(c) \equiv mp\_known$) {
    if ($rounding$) {
      int $cc = round\_unscaled(internal\_value(c))$;

      $print\_with\_leading\_zeroes(cc, ff)$;
    }
    else {
      $print\_number(internal\_value(c))$;
    }
  }
}
char $*mp\_set\_output\_file\_name$(**MP** mp, **integer** c)
{
  char $*ss = \Lambda$;   /* filename extension proposal */
  char $*nn = \Lambda$;   /* temp string for str() */
  unsigned $old\_setting$;   /* previous $selector$ setting */
  size_t i;   /* indexes into $filename\_template$ */
  integer f;   /* field width */

  $str\_room(1024)$;
  if ($mp \rightarrow job\_name \equiv \Lambda$) $mp\_open\_log\_file(mp)$;
  if ($internal\_string(mp\_output\_template) \equiv \Lambda$) {
    char $*s$;   /* a file extension derived from c */

    if ($c < 0$) $s = xstrdup(\texttt{".ps"})$;
    else ⟨ Use c to compute the file extension s 1252 ⟩;
    $mp\_pack\_job\_name(mp, s)$;
    $free(s)$;
    $ss = xstrdup(mp \rightarrow name\_of\_file)$;
  }
  else {   /* initializations */
    mp_string s, n, $ftemplate$;   /* a file extension derived from c */
    mp_number $saved\_char\_code$;

    $new\_number(saved\_char\_code)$;
    $number\_clone(saved\_char\_code, internal\_value(mp\_char\_code))$;
    $set\_internal\_from\_number(mp\_char\_code, unity\_t)$;
    $number\_multiply\_int(internal\_value(mp\_char\_code), c)$;
    if ($internal\_string(mp\_job\_name) \equiv \Lambda$) {
      if ($mp \rightarrow job\_name \equiv \Lambda$) {
        $mp \rightarrow job\_name = xstrdup(\texttt{"mpout"})$;
      }
      ⟨ Fix up $mp \rightarrow internal[mp\_job\_name]$ 868 ⟩;
    }
    $old\_setting = mp \rightarrow selector$;
    $mp \rightarrow selector = new\_string$;
    $i = 0$;
    $n = mp\_rts(mp, \texttt{""})$;   /* initialize */
    $ftemplate = internal\_string(mp\_output\_template)$;

while $(i < ftemplate \rightarrow len)$ {
  $f = 0$;
  if $(*(ftemplate \rightarrow str + i) \equiv$ '%') {
  CONTINUE: $incr(i)$;
    if $(i < ftemplate \rightarrow len)$ {
      switch $(*(ftemplate \rightarrow str + i))$ {
      case 'j': $mp\_append\_to\_template(mp, f, mp\_job\_name, true)$;
        break;
      case 'c':
        if $(number\_negative(internal\_value(mp\_char\_code)))$ {
          $mp\_print(mp, "ps")$;
        }
        else {
          $mp\_append\_to\_template(mp, f, mp\_char\_code, true)$;
        }
        break;
      case 'o': $mp\_append\_to\_template(mp, f, mp\_output\_format, true)$;
        break;
      case 'd': $mp\_append\_to\_template(mp, f, mp\_day, true)$;
        break;
      case 'm': $mp\_append\_to\_template(mp, f, mp\_month, true)$;
        break;
      case 'y': $mp\_append\_to\_template(mp, f, mp\_year, true)$;
        break;
      case 'H': $mp\_append\_to\_template(mp, f, mp\_hour, true)$;
        break;
      case 'M': $mp\_append\_to\_template(mp, f, mp\_minute, true)$;
        break;
      case '{':
        {   /∗ look up a name ∗/
          size_t $l = 0$;
          size_t $frst = i + 1$;
          while $(i < ftemplate \rightarrow len)$ {
            $i{+}{+}$;
            if $(*(ftemplate \rightarrow str + i) \equiv$ '}') **break**;
            $l{+}{+}$;
          }
          if $(l > 0)$ {
            mp_sym $p = mp\_id\_lookup(mp, ($**char** $*)(ftemplate \rightarrow str + frst), l, false)$;
            char $*id = xmalloc((l + 1), 1)$;
            (**void**) $memcpy(id, ($**char** $*)(ftemplate \rightarrow str + frst), ($**size_t**$) l)$;
            $*(id + l) =$ '\0';
            if $(p \equiv \Lambda)$ {
              char $err[256]$;
              $mp\_snprintf(err, 256,$
                 "requested␣identifier␣(%s)␣in␣outputtemplate␣not␣found.", $id)$;
              $mp\_warn(mp, err)$;
            }
            else {
              if $(eq\_type(p) \equiv mp\_internal\_quantity)$ {
                if $(equiv(p) \equiv mp\_output\_template)$ {

```
                        char err[256];
                        mp_snprintf (err, 256, "The␣appearance␣of␣outputtemplate␣inside\
                            ␣outputtemplate␣is␣ignored.");
                        mp_warn(mp, err);
                      }
                      else {
                        mp_append_to_template(mp, f, equiv(p), false);
                      }
                    }
                    else {
                      char err[256];
                      mp_snprintf (err, 256,
                          "requested␣identifier␣(%s)␣in␣outputtemplate␣is␣not␣an␣internal.",
                          id);
                      mp_warn(mp, err);
                    }
                  }
                  free(id);
                }
              }
              break;
            case '0': case '1': case '2': case '3': case '4': case '5': case '6': case '7':
              case '8': case '9':
              if ((f < 10)) f = (f * 10) + ftemplate→str[i] − '0';
              goto CONTINUE;
              break;
            case '%': mp_print_char(mp, '%');
              break;
            default:
              {
                char err[256];
                mp_snprintf (err, 256, "requested␣format␣(%c)␣in␣outputtemplate␣is␣unknown.",
                    *(ftemplate→str + i));
                mp_warn(mp, err);
              }
              mp_print_char(mp, *(ftemplate→str + i));
            }
          }
        }
        else {
          if (*(ftemplate→str + i) ≡ '.')
            if (n→len ≡ 0) n = mp_make_string(mp);
          mp_print_char(mp, *(ftemplate→str + i));
        }
        ;
        incr(i);
      }
      s = mp_make_string(mp);
      number_clone(internal_value(mp_char_code), saved_char_code);
      free_number(saved_char_code);
      mp→selector = old_setting;
```

```
    if (n→len ≡ 0) {
      n = s;
      s = mp_rts(mp, "");
    }
    ss = mp_str(mp, s);
    nn = mp_str(mp, n);
    mp_pack_file_name(mp, nn, "", ss);
    delete_str_ref(n);
    delete_str_ref(s);
  }
  return ss;
}

char *mp_get_output_file_name(MP mp)
{
  char *f;
  char *saved_name;       /* saved name_of_file */

  saved_name = xstrdup(mp→name_of_file);
  (void) mp_set_output_file_name(mp, round_unscaled(internal_value(mp_char_code)));
  f = xstrdup(mp→name_of_file);
  mp_pack_file_name(mp, saved_name, Λ, Λ);
  free(saved_name);
  return f;
}

void mp_open_output_file(MP mp)
{
  char *ss;       /* filename extension proposal */
  int c;       /* charcode rounded to the nearest integer */

  c = round_unscaled(internal_value(mp_char_code));
  ss = mp_set_output_file_name(mp, c);
  while (¬mp_open_out(mp, (void *) &mp→output_file, mp_filetype_postscript))
    mp_prompt_file_name(mp, "file␣name␣for␣output", ss);
  mp_store_true_output_filename(mp, c);
}
```

1252.   The file extension created here could be up to five characters long in extreme cases so it may have to be shortened on some systems.

⟨ Use c to compute the file extension s 1252 ⟩ ≡
```
  {
    s = xmalloc(7, 1);
    mp_snprintf(s, 7, ".%i", (int) c);
  }
```
This code is used in section 1251.

1253.   The user won't want to see all the output file names so we only save the first and last ones and a count of how many there were. For this purpose files are ordered primarily by **charcode** and secondarily by order of creation.

⟨ Internal library declarations 10 ⟩ +≡
```
  void mp_store_true_output_filename(MP mp, int c);
```

1254.   **void** *mp_store_true_output_filename*(**MP** *mp*, **int** *c*)
{
   if ((*c* < *mp*→*first_output_code*) ∧ (*mp*→*first_output_code* ≥ 0)) {
      mp→*first_output_code* = *c*;
      xfree(*mp*→*first_file_name*);
      mp→*first_file_name* = *xstrdup*(*mp*→*name_of_file*);
   }
   if (*c* ≥ *mp*→*last_output_code*) {
      mp→*last_output_code* = *c*;
      xfree(*mp*→*last_file_name*);
      mp→*last_file_name* = *xstrdup*(*mp*→*name_of_file*);
   }
   set_internal_string(*mp_output_filename*, *mp_rts*(*mp*, *mp*→*name_of_file*));
}

1255.   ⟨Global variables 14⟩ +≡
   char **first_file_name*;
   char **last_file_name*;   /* full file names */
   integer *first_output_code*;
   integer *last_output_code*;   /* rounded **charcode** values */
   integer *total_shipped*;   /* total number of *ship_out* operations completed */

1256.   ⟨Set initial values of key variables 38⟩ +≡
   mp→*first_file_name* = *xstrdup*("");
   mp→*last_file_name* = *xstrdup*("");
   mp→*first_output_code* = 32768;
   mp→*last_output_code* = −32768;
   mp→*total_shipped* = 0;

1257.   ⟨Dealloc variables 27⟩ +≡
   xfree(*mp*→*first_file_name*);
   xfree(*mp*→*last_file_name*);

1258.   ⟨Begin the progress report for the output of picture *c* 1258⟩ ≡
   if ((**int**) *mp*→*term_offset* > *mp*→*max_print_line* − 6) *mp_print_ln*(*mp*);
   else if ((*mp*→*term_offset* > 0) ∨ (*mp*→*file_offset* > 0)) *mp_print_char*(*mp*, *xord*('␣'));
   mp_print_char(*mp*, *xord*('[')); **if** (*c* ≥ 0) *mp_print_int*(*mp*, *c*)
This code is used in section 1273.

1259.   ⟨End progress report 1259⟩ ≡
   mp_print_char(*mp*, *xord*(']'));
   update_terminal( ); *incr*(*mp*→*total_shipped*)
This code is used in section 1273.

1260.   ⟨Explain what output files were written 1260⟩ ≡

    if (mp→*total_shipped* > 0) {
        mp_print_nl(mp, "");
        mp_print_int(mp, mp→*total_shipped*);
        if (mp→*noninteractive*) {
            mp_print(mp, "␣figure");
            if (mp→*total_shipped* > 1) *mp_print_char*(mp, xord('s'));
            mp_print(mp, "␣created.");
        }
        else {
            mp_print(mp, "␣output␣file");
            if (mp→*total_shipped* > 1) *mp_print_char*(mp, xord('s'));
            mp_print(mp, "␣written:␣");
            mp_print(mp, mp→*first_file_name*);
            if (mp→*total_shipped* > 1) {
                if (31 + *strlen*(mp→*first_file_name*) + *strlen*(mp→*last_file_name*) > (**unsigned**) mp→*max_print_line*)
                    mp_print_ln(mp);
                mp_print(mp, "␣..␣");
                mp_print(mp, mp→*last_file_name*);
            }
            mp_print_nl(mp, "");
        }
    }

This code is used in section 1287.

1261.   ⟨Internal library declarations 10⟩ +≡
    boolean *mp_has_font_size*(**MP** mp, **font_number** f);

1262.   **boolean** *mp_has_font_size*(**MP** mp, **font_number** f)
    {
        return (mp→*font_sizes*[f] ≠ Λ);
    }

1263.   The **special** command saves up lines of text to be printed during the next *ship_out* operation. The saved items are stored as a list of capsule tokens.

⟨Global variables 14⟩ +≡
    mp_node *last_pending*;   /* the last token in a list of pending specials */

1264.   ⟨Declare action procedures for use by *do_statement* 1048⟩ +≡
    static void *mp_do_special*(**MP** mp);

1265.   **void** $mp\_do\_special(\textbf{MP}\ mp)$

$\{$
   $mp\_get\_x\_next(mp);$
   $mp\_scan\_expression(mp);$
   if $(mp\text{-}cur\_exp.type \neq mp\_string\_type)$ $\{$
     $\langle$ Complain about improper special operation 1266 $\rangle;$
   $\}$
   else $\{$
     $mp\_link(mp\text{-}last\_pending) = mp\_stash\_cur\_exp(mp);$
     $mp\text{-}last\_pending = mp\_link(mp\text{-}last\_pending);$
     $mp\_link(mp\text{-}last\_pending) = \Lambda;$
   $\}$
$\}$

1266. $\langle$ Complain about improper special operation 1266 $\rangle \equiv$

$\{$
   const char $*hlp[\,] = \{$"$\texttt{Only}_\sqcup\texttt{known}_\sqcup\texttt{strings}_\sqcup\texttt{are}_\sqcup\texttt{allowed}_\sqcup\texttt{for}_\sqcup\texttt{output}_\sqcup\texttt{as}_\sqcup\texttt{specials.}$", $\Lambda\};$
   $mp\_disp\_err(mp, \Lambda);$
   $mp\_back\_error(mp,$ "$\texttt{Unsuitable}_\sqcup\texttt{expression}$", $hlp, true);$
   $mp\_get\_x\_next(mp);$
$\}$

This code is used in section 1265.

1267.   On the export side, we need an extra object type for special strings.

$\langle$ Graphical object codes 459 $\rangle +\equiv$
   $mp\_special\_code = 8$,

1268. $\langle$ Export pending specials 1268 $\rangle \equiv$
   $p = mp\_link(mp\text{-}spec\_head);$ **while** $(p \neq \Lambda)$ $\{$ $mp\_special\_object * tp;$ $tp = (\ mp\_special\_object\ *\ )$
     $mp\_new\_graphic\_object(mp, mp\_special\_code);$
   $gr\_pre\_script(tp) = mp\_xstrdup(mp, mp\_str(mp, value\_str(p)));$ **if** $(hh\text{-}body \equiv \Lambda)$ $hh\text{-}body = ($
     $mp\_graphic\_object *)\ tp;$ **else** $gr\_link(hp) = (\ mp\_graphic\_object\ *\ )\ tp;$ $hp = (\ mp\_graphic\_object\ *$
     $)\ tp;$
   $p = mp\_link(p);$ $\}$ $mp\_flush\_token\_list(mp, mp\_link(mp\text{-}spec\_head));$
   $mp\_link(mp\text{-}spec\_head) = \Lambda;$ $mp\text{-}last\_pending = mp\text{-}spec\_head$

This code is used in section 1270.

1269.   We are now ready for the main output procedure. Note that the *selector* setting is saved in a global variable so that *begin_diagnostic* can access it.

$\langle$ Declare the PostScript output procedures 1269 $\rangle \equiv$
   static void $mp\_ship\_out(\textbf{MP}\ mp, \textbf{mp\_node}\ h);$

This code is used in section 1137.

1270.   Once again, the *gr_XXXX* macros are defined in *mppsout.h*

#define   *export_color*(*q, p*)
        if (**mp_color_model**(*p*) ≡ *mp_uninitialized_model*) {
          gr_color_model(*q*)   =   (**unsigned**
              char)(*number_to_scaled*(*internal_value*(*mp_default_color_model*))/65536);
          gr_cyan_val(*q*) = 0;
          gr_magenta_val(*q*) = 0;
          gr_yellow_val(*q*) = 0;
          gr_black_val(*q*) = ((*gr_color_model*(*q*) ≡ *mp_cmyk_model* ? *number_to_scaled*(*unity_t*) :
            0)/65536.0);
        }
        else {
          gr_color_model(*q*) = (**unsigned char**) **mp_color_model**(*p*);
          gr_cyan_val(*q*) = *number_to_double*(*p⃗cyan*);
          gr_magenta_val(*q*) = *number_to_double*(*p⃗magenta*);
          gr_yellow_val(*q*) = *number_to_double*(*p⃗yellow*);
          gr_black_val(*q*) = *number_to_double*(*p⃗black*);
        }
#define   *export_scripts*(*q, p*)
          if (*mp_pre_script*(*p*) ≠ Λ) *gr_pre_script*(*q*) = *mp_xstrdup*(*mp, mp_str*(*mp, mp_pre_script*(*p*)));
          if (*mp_post_script*(*p*) ≠ Λ) *gr_post_script*(*q*) = *mp_xstrdup*(*mp, mp_str*(*mp, mp_post_script*(*p*)));
  struct *mp_edge_object* *∗mp_gr_export*(**MP** *mp,* **mp_edge_header_node** *h*){ **mp_node** *p*;
        /∗ the current graphical object ∗/
      integer *t*;   /∗ a temporary value ∗/
      integer *c*;   /∗ a rounded charcode ∗/
      mp_number *d_width*;   /∗ the current pen width ∗/
      mp_edge_object ∗ *hh*;   /∗ the first graphical object ∗/
      mp_graphic_object ∗ *hq*;   /∗ something *hp* points to ∗/
      mp_text_object ∗ *tt*;
      mp_fill_object ∗ *tf*;
      mp_stroked_object ∗ *ts*;
      mp_clip_object ∗ *tc*;
      mp_bounds_object ∗ *tb*;
      mp_graphic_object ∗ *hp* = Λ;   /∗ the current graphical object ∗/
      mp_set_bbox(*mp, h, true*);
      hh = *xmalloc*(1, **sizeof** (*mp_edge_object*));
      hh⃗body = Λ;
      hh⃗next = Λ;
      hh⃗parent = *mp*;
      hh⃗minx = *number_to_double*(*h⃗minx*);
      hh⃗minx = (*fabs*(*hh⃗minx*) < 0.00001 ? 0 : *hh⃗minx*);
      hh⃗miny = *number_to_double*(*h⃗miny*);
      hh⃗miny = (*fabs*(*hh⃗miny*) < 0.00001 ? 0 : *hh⃗miny*);
      hh⃗maxx = *number_to_double*(*h⃗maxx*);
      hh⃗maxx = (*fabs*(*hh⃗maxx*) < 0.00001 ? 0 : *hh⃗maxx*);
      hh⃗maxy = *number_to_double*(*h⃗maxy*);
      hh⃗maxy = (*fabs*(*hh⃗maxy*) < 0.00001 ? 0 : *hh⃗maxy*);
      hh⃗filename = *mp_get_output_file_name*(*mp*);
      c = *round_unscaled*(*internal_value*(*mp_char_code*));
      hh⃗charcode = *c*;
      hh⃗width = *number_to_double*(*internal_value*(*mp_char_wd*));

hh→$height = number\_to\_double(internal\_value(mp\_char\_ht));$
hh→$depth = number\_to\_double(internal\_value(mp\_char\_dp));$
hh→$ital\_corr = number\_to\_double(internal\_value(mp\_char\_ic));$
⟨ Export pending specials 1268 ⟩;
$p = mp\_link(edge\_list(h));$ **while** $(p \neq \Lambda)$ { $hq = mp\_new\_graphic\_object(mp,$
$\quad$ (**int**)$((mp\_type(p) - mp\_fill\_node\_type) + 1));$ **switch** $(mp\_type(p))$ { **case** $mp\_fill\_node\_type$: {
$\quad$ **mp_fill_node** $p0 = ($**mp_fill_node**$)$ p; $tf = ($ $mp\_fill\_object *$ $)$ hq;
$gr\_pen\_p(tf) = mp\_export\_knot\_list(mp, mp\_pen\_p(p0));$
$new\_number(d\_width);$
$mp\_get\_pen\_scale(mp, \&d\_width, mp\_pen\_p(p0));$ $\qquad$ /* whats the point ? */
$free\_number(d\_width);$
if $((mp\_pen\_p(p0) \equiv \Lambda) \vee pen\_is\_elliptical(mp\_pen\_p(p0)))$ {
$\quad gr\_path\_p(tf) = mp\_export\_knot\_list(mp, mp\_path\_p(p0));$
}
else {
$\quad$ **mp_knot** pc, pp;

$\quad pc = mp\_copy\_path(mp, mp\_path\_p(p0));$
$\quad pp = mp\_make\_envelope(mp, pc, mp\_pen\_p(p0), p0$→$ljoin, 0, p0$→$miterlim);$
$\quad gr\_path\_p(tf) = mp\_export\_knot\_list(mp, pp);$
$\quad mp\_toss\_knot\_list(mp, pp);$
$\quad pc = mp\_htap\_ypoc(mp, mp\_path\_p(p0));$
$\quad pp = mp\_make\_envelope(mp, pc, mp\_pen\_p(($**mp_fill_node**$)$ $p), p0$→$ljoin, 0, p0$→$miterlim);$
$\quad gr\_htap\_p(tf) = mp\_export\_knot\_list(mp, pp);$
$\quad mp\_toss\_knot\_list(mp, pp);$
}
$export\_color(tf, p0);$
$export\_scripts(tf, p);$
$gr\_ljoin\_val(tf) = p0$→$ljoin;$
$gr\_miterlim\_val(tf) = number\_to\_double(p0$→$miterlim);$ } **break**; **case** $mp\_stroked\_node\_type$: {
$\quad$ **mp_stroked_node** $p0 = ($**mp_stroked_node**$)$ p; $ts = ($ $mp\_stroked\_object *$ $)$ hq;
$gr\_pen\_p(ts) = mp\_export\_knot\_list(mp, mp\_pen\_p(p0));$
$new\_number(d\_width);$
$mp\_get\_pen\_scale(mp, \&d\_width, mp\_pen\_p(p0));$
if $(pen\_is\_elliptical(mp\_pen\_p(p0)))$ {
$\quad gr\_path\_p(ts) = mp\_export\_knot\_list(mp, mp\_path\_p(p0));$
}
else {
$\quad$ **mp_knot** pc;

$\quad pc = mp\_copy\_path(mp, mp\_path\_p(p0));$
$\quad t = p0$→$lcap;$
$\quad$ **if** $(mp\_left\_type(pc) \neq mp\_endpoint)$ {
$\quad\quad mp\_left\_type(mp\_insert\_knot(mp, pc, pc$→$x\_coord, pc$→$y\_coord)) = mp\_endpoint;$
$\quad\quad mp\_right\_type(pc) = mp\_endpoint;$
$\quad\quad pc = mp\_next\_knot(pc);$
$\quad\quad t = 1;$
$\quad$ }
$\quad pc = mp\_make\_envelope(mp, pc, mp\_pen\_p(p0), p0$→$ljoin, ($**quarterword**$)$ $t, p0$→$miterlim);$
$\quad gr\_path\_p(ts) = mp\_export\_knot\_list(mp, pc);$
$\quad mp\_toss\_knot\_list(mp, pc);$
}
$export\_color(ts, p0);$
$export\_scripts(ts, p);$

$gr\_ljoin\_val(ts) = p0 \rightarrow ljoin;$

$gr\_miterlim\_val(ts) = number\_to\_double(p0 \rightarrow miterlim);$

$gr\_lcap\_val(ts) = p0 \rightarrow lcap;$

$gr\_dash\_p(ts) = mp\_export\_dashes(mp, p0, d\_width);$

$free\_number(d\_width);$ } **break**; **case** $mp\_text\_node\_type$: { **mp_text_node** $p0 = ($**mp_text_node**$)$
    $p;$ $tt = ($ $mp\_text\_object *)$ $hq;$

$gr\_text\_p(tt) = mp\_xstrldup(mp, mp\_str(mp, mp\_text\_p(p)), mp\_text\_p(p) \rightarrow len);$

$gr\_text\_l(tt) = ($**size_t**$)$ $mp\_text\_p(p) \rightarrow len;$

$gr\_font\_n(tt) = ($**unsigned int**$)$ $mp\_font\_n(p);$

$gr\_font\_name(tt) = mp\_xstrdup(mp, mp \rightarrow font\_name[mp\_font\_n(p)]);$

$gr\_font\_dsize(tt) = mp \rightarrow font\_dsize[mp\_font\_n(p)]/65536.0;$

$export\_color(tt, p0);$

$export\_scripts(tt, p);$

$gr\_width\_val(tt) = number\_to\_double(p0 \rightarrow width);$

$gr\_height\_val(tt) = number\_to\_double(p0 \rightarrow height);$

$gr\_depth\_val(tt) = number\_to\_double(p0 \rightarrow depth);$

$gr\_tx\_val(tt) = number\_to\_double(p0 \rightarrow tx);$

$gr\_ty\_val(tt) = number\_to\_double(p0 \rightarrow ty);$

$gr\_txx\_val(tt) = number\_to\_double(p0 \rightarrow txx);$

$gr\_txy\_val(tt) = number\_to\_double(p0 \rightarrow txy);$

$gr\_tyx\_val(tt) = number\_to\_double(p0 \rightarrow tyx);$

$gr\_tyy\_val(tt) = number\_to\_double(p0 \rightarrow tyy);$ } **break**; **case** $mp\_start\_clip\_node\_type$: $tc = ($
    $mp\_clip\_object *)$ $hq;$

$gr\_path\_p(tc) = mp\_export\_knot\_list(mp, mp\_path\_p(($**mp_start_clip_node**$)$ $p));$

break; **case** $mp\_start\_bounds\_node\_type$: $tb = ($ $mp\_bounds\_object *)$ $hq;$

$gr\_path\_p(tb) = mp\_export\_knot\_list(mp, mp\_path\_p(($**mp_start_bounds_node**$)$ $p));$

break;

case $mp\_stop\_clip\_node\_type$: **case** $mp\_stop\_bounds\_node\_type$:   /* nothing to do here */

break;

default: /* there are no other valid cases, but please the compiler */

break; }

if $(hh \rightarrow body \equiv \Lambda)$ $hh \rightarrow body = hq;$

else $gr\_link(hp) = hq;$

$hp = hq;$

$p = mp\_link(p);$ } **return** $hh;$ }

1271.   This function is only used for the *glyph* operator, so it takes quite a few shortcuts for cases that cannot appear in the output of *mp_ps_font_charstring*.

mp_edge_header_node *mp_gr_import*(**MP** *mp*, **struct** *mp_edge_object* *∗hh*){ **mp_edge_header_node**
        h;   /∗ the edge object ∗/
    mp_node *ph*, *pn*, *pt*;   /∗ for adding items ∗/
    mp_graphic_object ∗ p;   /∗ the current graphical object ∗/
    h = *mp_get_edge_header_node*(*mp*);
    mp_init_edges(*mp*, *h*);
    ph = *edge_list*(*h*);
    pt = *ph*;
    p = *hh→body*;
    set_number_from_double(*h→minx*, *hh→minx*);
    set_number_from_double(*h→miny*, *hh→miny*);
    set_number_from_double(*h→maxx*, *hh→maxx*);
    set_number_from_double(*h→maxy*, *hh→maxy*); **while** (*p* ≠ Λ) { **switch** (*gr_type*(*p*)) { **case**
        mp_fill_code: **if** (*gr_pen_p* ((*mp_fill_object* ∗) *p*) ≡ Λ) { **mp_number** *turns*;

    new_number(*turns*);
    pn = *mp_new_fill_node*(*mp*, Λ); *mp_path_p*((**mp_fill_node**) *pn*) = *mp_import_knot_list* (*mp*,
        gr_path_p ((*mp_fill_object* ∗) *p*)) ;
    mp_color_model(*pn*) = *mp_grey_model*;
    mp_turn_cycles(*mp*, &*turns*, *mp_path_p*((**mp_fill_node**) *pn*));
    if (*number_negative*(*turns*)) {
      set_number_to_unity(((**mp_fill_node**) *pn*)→*grey*);
      mp_link(*pt*) = *pn*;
      pt = *mp_link*(*pt*);
    }
    else {
      set_number_to_zero(((**mp_fill_node**) *pn*)→*grey*);
      mp_link(*pn*) = *mp_link*(*ph*);
      mp_link(*ph*) = *pn*;
      if (*ph* ≡ *pt*) *pt* = *pn*;
    }
    free_number(*turns*); } **break**;
  case *mp_stroked_code*: **case** *mp_text_code*: **case** *mp_start_clip_code*: **case** *mp_stop_clip_code*:
    case *mp_start_bounds_code*: **case** *mp_stop_bounds_code*: **case** *mp_special_code*: **break**; }
      /∗ all cases are enumerated ∗/
    p = *p→next*; } *mp_gr_toss_objects*(*hh*);
    return *h*; }

1272.   ⟨ Declarations 8 ⟩ +≡
  struct *mp_edge_object* *∗mp_gr_export*(**MP** *mp*, **mp_edge_header_node** *h*);
  mp_edge_header_node *mp_gr_import*(**MP** *mp*, **struct** *mp_edge_object* *∗h*);

1273.   This function is now nearly trivial.

> **void** $mp\_ship\_out(\mathbf{MP}\ mp, \mathbf{mp\_node}\ h)$
> {   /∗ output edge structure h ∗/
>   **int** c;   /∗ **charcode** rounded to the nearest integer ∗/
> $c = round\_unscaled(internal\_value(mp\_char\_code))$;
>   ⟨Begin the progress report for the output of picture c 1258⟩;
> $(mp\text{→}shipout\_backend)(mp, h)$;
>   ⟨End progress report 1259⟩;
>   **if** $(number\_positive(internal\_value(mp\_tracing\_output)))$
> $mp\_print\_edges(mp, h, "_⊔(just_⊔shipped_⊔out)", true)$;
> }

1274.   ⟨Declarations 8⟩ +≡
  static void $mp\_shipout\_backend(\mathbf{MP}\ mp, \mathbf{void}\ *h)$;

1275.

> **void** $mp\_shipout\_backend(\mathbf{MP}\ mp, \mathbf{void}\ *voidh)$
> {
>   **char** $*s$;
> $mp\_edge\_object * hh$;   /∗ the first graphical object ∗/
>   **mp_edge_header_node** $h = (\mathbf{mp\_edge\_header\_node})\ voidh$;
> $hh = mp\_gr\_export(mp, h)$;
> $s = \Lambda$;
>   **if** $(internal\_string(mp\_output\_format) \neq \Lambda)\ s = mp\_str(mp, internal\_string(mp\_output\_format))$;
>   **if** $(s \wedge strcmp(s, "svg") \equiv 0)$ {
> $(\mathbf{void})\ mp\_svg\_gr\_ship\_out(hh, (number\_to\_scaled(internal\_value(mp\_prologues)))/65536, false)$;
>   }
>   **else if** $(s \wedge strcmp(s, "png") \equiv 0)$ {
> $(\mathbf{void})\ mp\_png\_gr\_ship\_out(hh, (\mathbf{const\ char}\ *)((internal\_string(mp\_output\_format\_options))\text{→}str),$
> $false)$;
>   }
>   **else** {
> $(\mathbf{void})\ mp\_gr\_ship\_out(hh, (number\_to\_scaled(internal\_value(mp\_prologues)))/65536,$
> $(number\_to\_scaled(internal\_value(mp\_procset)))/65536, false)$;
>   }
> $mp\_gr\_toss\_objects(hh)$;
> }

1276.   ⟨Exported types 15⟩ +≡
  typedef void$(*mp\_backend\_writer)(\mathbf{MP}, \mathbf{void}\ *)$;

1277.   ⟨Option variables 26⟩ +≡
  $mp\_backend\_writer\ shipout\_backend$;

1278.   Now that we've finished *ship_out*, let's look at the other commands by which a user can send things to the GF file.

1279.   ⟨Global variables 14⟩ +≡
  psout_data ps;
  svgout_data svg;
  pngout_data png;

1280. ⟨ Allocate or initialize variables 28 ⟩ +≡
  $mp\_ps\_backend\_initialize\,(mp)$;
  $mp\_svg\_backend\_initialize\,(mp)$;
  $mp\_png\_backend\_initialize\,(mp)$;

1281. ⟨ Dealloc variables 27 ⟩ +≡
  $mp\_ps\_backend\_free\,(mp)$;
  $mp\_svg\_backend\_free\,(mp)$;
  $mp\_png\_backend\_free\,(mp)$;

1282.   Dumping and undumping the tables.
When MP is started, it is possible to preload a macro file containing definitions that will be usable in the main input file. This action even takes place automatically, based on the name of the executable (mpost will attempt to preload the macros in the file mpost.mp). If such a preload is not desired, the option variable *ini_version* has to be set *true*.

The variable *mem_file* holds the open file pointer.

⟨ Global variables 14 ⟩ +≡
  void *mem_file*;   /∗ file for input or preloaded macros ∗/

1283.   ⟨ Declarations 8 ⟩ +≡
  extern boolean *mp_load_preload_file*(**MP** *mp*);

1284.   Preloading a file is a lot like *mp_run* itself, except that METAPOST should not exit and that a bit of trickery is needed with the input buffer to make sure that the preloading does not interfere with the actual job.

```
boolean mp_load_preload_file(MP mp){ size_t k;
    in_state_record old_state;
    integer old_in_open = mp→in_open;
    void *old_cur_file = cur_file;
    char *fname = xstrdup(mp→name_of_file);
    size_t l = strlen(fname);

    old_state = mp→cur_input;
    str_room(l);
    for (k = 0; k < l; k++) {
        append_char(*(fname + k));
    }
    name = mp_make_string(mp);
    if (¬mp→log_opened) {
        mp_open_log_file(mp);
    }      /* open_log_file doesn't show_context, so limit and loc needn't be set to meaningful values yet
           */
    if (((int) mp→term_offset + (int) strlen(fname)) > (mp→max_print_line − 2)) mp_print_ln(mp);
    else if ((mp→term_offset > 0) ∨ (mp→file_offset > 0)) mp_print_char(mp, xord(' '));
    mp_print_char(mp, xord('('));
    incr(mp→open_parens);
    mp_print(mp, fname);
    update_terminal(); { line = 1;
    start = loc = limit + (mp→noninteractive ? 0 : 1);
    cur_file = mp→mem_file;
    (void) mp_input_ln(mp, cur_file);
    mp_firm_up_the_line(mp);
    mp→buffer[limit] = xord('%');
    mp→first = (size_t)(limit + 1);
    loc = start; } mp→reading_preload = true;
    do {
        mp_do_statement(mp);
    } while (¬(cur_cmd() ≡ mp_stop));      /* "dump" or EOF */
    mp→reading_preload = false;
    mp_primitive(mp, "dump", mp_relax, 0);      /* reset dump */
    while (mp→input_ptr > 0) {
        if (token_state) mp_end_token_list(mp);
        else mp_end_file_reading(mp);
    }
    while (mp→loop_ptr ≠ Λ) mp_stop_iteration(mp);
    while (mp→open_parens > 0) {
        mp_print(mp, " )");
        decr(mp→open_parens);
    }
    ;
    while (mp→cond_ptr ≠ Λ) {
        mp_print_nl(mp, "(dump occurred when ");
        ;
        mp_print_cmd_mod(mp, mp_fi_or_else, mp→cur_if);      /* 'if' or 'elseif' or 'else' */
        if (mp→if_line ≠ 0) {
```

```
    mp_print(mp, "␣on␣line␣");
    mp_print_int(mp, mp→if_line);
  }
  mp_print(mp, "␣was␣incomplete)");
  mp→if_line = if_line_field(mp→cond_ptr);
  mp→cur_if = mp_name_type(mp→cond_ptr);
  mp→cond_ptr = mp_link(mp→cond_ptr);
}    /* (mp→close_file)(mp, mp→mem_file); */
cur_file = old_cur_file;
mp→cur_input = old_state;
mp→in_open = old_in_open;
return true; }
```

1285.   The main program.   This is it: the part of METAPOST that executes all those procedures we have written.

Well—almost. We haven't put the parsing subroutines into the program yet; and we'd better leave space for a few more routines that may have been forgotten.

⟨ Declare the basic parsing subroutines 931 ⟩;
⟨ Declare miscellaneous procedures that were declared *forward* 247 ⟩

1286.   Here we do whatever is needed to complete METAPOST's job gracefully on the local operating system. The code here might come into play after a fatal error; it must therefore consist entirely of "safe" operations that cannot produce error messages. For example, it would be a mistake to call *str_room* or *make_string* at this time, because a call on *overflow* might lead to an infinite loop.

1287.   void *mp_close_files_and_terminate*(**MP** *mp*)
{
   integer *k*;   /∗ all-purpose index ∗/
   integer LH;   /∗ the length of the TFM header, in words ∗/
   int *lk_offset*;   /∗ extra words inserted at beginning of *lig_kern* array ∗/
   mp_node *p*;   /∗ runs through a list of TFM dimensions ∗/

   if (*mp→finished*) **return**;
   ⟨ Close all open files in the *rd_file* and *wr_file* arrays 1289 ⟩;
   if (*number_positive*(*internal_value*(*mp_tracing_stats*))) ⟨ Output statistics about this job 1292 ⟩;
   wake_up_terminal( );
   ⟨ Do all the finishing work on the TFM file 1291 ⟩;
   ⟨ Explain what output files were written 1260 ⟩;
   if (*mp→log_opened* ∧ ¬*mp→noninteractive*) {
     wlog_cr;
     (*mp→close_file*)(*mp*, *mp→log_file*);
     mp→selector = *mp→selector* − 2;
     if (*mp→selector* ≡ *term_only*) {
       mp_print_nl(*mp*, "Transcript␣written␣on␣");
       ;
       mp_print(*mp*, *mp→log_name*);
       mp_print_char(*mp*, *xord*('.'));
     }
   }
   mp_print_ln(*mp*);
   mp→finished = *true*;
}

1288.   ⟨ Declarations 8 ⟩ +≡
  static void *mp_close_files_and_terminate*(**MP** *mp*);

1289.   ⟨Close all open files in the *rd_file* and *wr_file* arrays 1289⟩ ≡
  if (*mp*→*rd_fname* ≠ Λ) {
    for (*k* = 0; *k* < (**int**) *mp*→*read_files*; *k*++) {
      if (*mp*→*rd_fname*[*k*] ≠ Λ) {
        (*mp*→*close_file*)(*mp*, *mp*→*rd_file*[*k*]);
        xfree(*mp*→*rd_fname*[*k*]);
      }
    }
  }
  if (*mp*→*wr_fname* ≠ Λ) {
    for (*k* = 0; *k* < (**int**) *mp*→*write_files*; *k*++) {
      if (*mp*→*wr_fname*[*k*] ≠ Λ) {
        (*mp*→*close_file*)(*mp*, *mp*→*wr_file*[*k*]);
        xfree(*mp*→*wr_fname*[*k*]);
      }
    }
  }

This code is used in section 1287.

1290.   ⟨Dealloc variables 27⟩ +≡
  for (*k* = 0; *k* < (**int**) *mp*→*max_read_files*; *k*++) {
    if (*mp*→*rd_fname*[*k*] ≠ Λ) {
      (*mp*→*close_file*)(*mp*, *mp*→*rd_file*[*k*]);
      xfree(*mp*→*rd_fname*[*k*]);
    }
  }
  xfree(*mp*→*rd_file*);
  xfree(*mp*→*rd_fname*);
  for (*k* = 0; *k* < (**int**) *mp*→*max_write_files*; *k*++) {
    if (*mp*→*wr_fname*[*k*] ≠ Λ) {
      (*mp*→*close_file*)(*mp*, *mp*→*wr_file*[*k*]);
      xfree(*mp*→*wr_fname*[*k*]);
    }
  }
  xfree(*mp*→*wr_file*);
  xfree(*mp*→*wr_fname*);

1291.   We want to produce a `TFM` file if and only if *mp_fontmaking* is positive.
  We reclaim all of the variable-size memory at this point, so that there is no chance of another memory overflow after the memory capacity has already been exceeded.

⟨Do all the finishing work on the `TFM` file 1291⟩ ≡
  if (*number_positive*(*internal_value*(*mp_fontmaking*))) {
    ⟨Massage the `TFM` widths 1204⟩;
    mp_fix_design_size(*mp*);
    mp_fix_check_sum(*mp*);
    ⟨Massage the `TFM` heights, depths, and italic corrections 1206⟩;
    set_number_to_zero(*internal_value*(*mp_fontmaking*));   /∗ avoid loop in case of fatal error ∗/
    ⟨Finish the `TFM` file 1217⟩;
  }

This code is used in section 1287.

1292.   The present section goes directly to the log file instead of using *print* commands, because there's no need for these strings to take up *str_pool* memory when a non-**stat** version of METAPOST is being used.

⟨Output statistics about this job 1292⟩ ≡

  if (*mp⃗log_opened*) {

    char *s*[128];

    wlog_ln("␣");
    wlog_ln("Here␣is␣how␣much␣of␣MetaPost's␣memory␣you␣used:");
    ;
    mp_snprintf(*s*, 128, "␣%i␣string%s␣using␣%i␣character%s", (**int**) *mp⃗max_strs_used*,
        (*mp⃗max_strs_used* ≠ 1 ? "s" : ""), (**int**) *mp⃗max_pl_used*, (*mp⃗max_pl_used* ≠ 1 ? "s" : ""));
    wlog_ln(*s*);
    mp_snprintf(*s*, 128, "␣%i␣bytes␣of␣node␣memory", (**int**) *mp⃗var_used_max*);
    wlog_ln(*s*);
    mp_snprintf(*s*, 128, "␣%i␣symbolic␣tokens", (**int**) *mp⃗st_count*);
    wlog_ln(*s*);
    mp_snprintf(*s*, 128, "␣%ii,%in,%ip,%ib,%if␣stack␣positions␣out␣of␣%ii,%in,%ip,%ib,%if", (**int**)
        mp⃗max_in_stack, (**int**) *mp⃗int_ptr*, (**int**) *mp⃗max_param_stack*, (**int**) *mp⃗max_buf_stack* + 1, (**int**)
        mp⃗in_open_max − *file_bottom*, (**int**) *mp⃗stack_size*, (**int**) *mp⃗max_internal*, (**int**)
        mp⃗param_size, (**int**) *mp⃗buf_size*, (**int**) *mp⃗max_in_open* − *file_bottom*);
    wlog_ln(*s*);

  }

This code is used in section 1287.

1293.   It is nice to have have some of the stats available from the API.

⟨Exported function headers 18⟩ +≡

  int *mp_memory_usage*(**MP** *mp*);
  int *mp_hash_usage*(**MP** *mp*);
  int *mp_param_usage*(**MP** *mp*);
  int *mp_open_usage*(**MP** *mp*);

1294.   **int** *mp_memory_usage*(**MP** *mp*)

  {

    return (**int**) *mp⃗var_used*;

  }

  int *mp_hash_usage*(**MP** *mp*)

  {

    return (**int**) *mp⃗st_count*;

  }

  int *mp_param_usage*(**MP** *mp*)

  {

    return (**int**) *mp⃗max_param_stack*;

  }

  int *mp_open_usage*(**MP** *mp*)

  {

    return (**int**) *mp⃗max_in_stack*;

  }

1295.   We get to the *final_cleanup* routine when **end** or **dump** has been scanned.

```
void mp_final_cleanup(MP mp)
{     /* -Wunused: integer c; */       /* 0 for end, 1 for dump */
      /* clang: never read: c = cur_mod(); */
   if (mp→job_name ≡ Λ) mp_open_log_file(mp);
   while (mp→input_ptr > 0) {
      if (token_state) mp_end_token_list(mp);
      else  mp_end_file_reading(mp);
   }
   while (mp→loop_ptr ≠ Λ)  mp_stop_iteration(mp);
   while (mp→open_parens > 0) {
      mp_print(mp, "␣)");
      decr(mp→open_parens);
   }
   ;
   while (mp→cond_ptr ≠ Λ) {
      mp_print_nl(mp, "(end␣occurred␣when␣");
      ;
      mp_print_cmd_mod(mp, mp_fi_or_else, mp→cur_if);       /* 'if' or 'elseif' or 'else' */
      if (mp→if_line ≠ 0) {
         mp_print(mp, "␣on␣line␣");
         mp_print_int(mp, mp→if_line);
      }
      mp_print(mp, "␣was␣incomplete)");
      mp→if_line = if_line_field(mp→cond_ptr);
      mp→cur_if = mp_name_type(mp→cond_ptr);
      mp→cond_ptr = mp_link(mp→cond_ptr);
   }
   if (mp→history ≠ mp_spotless)
      if (((mp→history ≡ mp_warning_issued) ∨ (mp→interaction < mp_error_stop_mode)))
         if (mp→selector ≡ term_and_log) {
            mp→selector = term_only;
            mp_print_nl(mp, "(see␣the␣transcript␣file␣for␣additional␣information)");
            ;
            mp→selector = term_and_log;
         }
}
```

1296.   ⟨Declarations 8⟩ +≡

```
static void mp_final_cleanup(MP mp);
static void mp_init_prim(MP mp);
static void mp_init_tab(MP mp);
```

1297.   void *mp_init_prim*(**MP** *mp*)

```
{     /* initialize all the primitives */
   ⟨Put each of METAPOST's primitives into the hash table 200⟩;
}
void mp_init_tab(MP mp)
{     /* initialize other tables */
   ⟨Initialize table entries 182⟩;
}
```

1298.   When we begin the following code, METAPOST's tables may still contain garbage; thus we must proceed cautiously to get bootstrapped in.

But when we finish this part of the program, METAPOST is ready to call on the *main_control* routine to do its work.

⟨ Get the first line of input and prepare to start 1298 ⟩ ≡
```
{
    ⟨ Initialize the input routines 717 ⟩;
    if (¬mp→ini_version) {
        if (¬mp_load_preload_file(mp)) {
            mp→history = mp_fatal_error_stop;
            return mp;
        }
    }
    ⟨ Initializations following first line 1299 ⟩;
}
```
This code is used in section 16.

1299.   ⟨ Initializations following first line 1299 ⟩ ≡
$mp{\rightarrow}buffer[\mathit{limit}] = (\textbf{ASCII\_code})\ \text{'\%'}$;
$mp\_fix\_date\_and\_time(mp)$;
if $(mp{\rightarrow}random\_seed \equiv 0)$
   $mp{\rightarrow}random\_seed = (number\_to\_scaled(internal\_value(mp\_time))/number\_to\_scaled(unity\_t)) +$
      $number\_to\_scaled(internal\_value(mp\_day))$;
$init\_randoms(mp{\rightarrow}random\_seed)$;
$initialize\_print\_selector()$;
$mp\_normalize\_selector(mp)$;
if $(loc < limit)$
   if $(mp{\rightarrow}buffer[loc] \neq \text{'\textbackslash\textbackslash'})\ mp\_start\_input(mp)$;   /* **input** assumed */
This code is used in section 1298.

1300.   Debugging.

1301.   System-dependent changes.   This section should be replaced, if necessary, by any special modification of the program that are necessary to make METAPOST work at a particular installation. It is usually best to design your change file so that all changes to previous sections preserve the section numbering; then everybody's version will be consistent with the published program. More extensive changes, which introduce new sections, can be inserted here; then only the index itself will get a new section number.

1302.   Index.   Here is where you can find all uses of each identifier in the program, with underlined entries pointing to where the identifier was defined. If the identifier is only one letter long, however, you get to see only the underlined entries. *All references are to section numbers instead of page numbers.*

This index also lists error messages and other aspects of the program that you might want to look up some day. For example, the entry for "system dependencies" lists all sections that should receive special attention from people who are installing METAPOST in a new operating environment. A list of various things that can't happen appears under "this can't happen". Approximately 25 sections are listed under "inner loop"; these account for more than 60% of METAPOST's running time, exclusive of input and output.

& primitive: 955.
!: 102.
* primitive: 955.
**: 69, 875.
+ primitive: 955.
++ primitive: 955.
+-+ primitive: 955.
, primitive: 232.
- primitive: 955.
->: 250.
.. primitive: 232.
/ primitive: 955.
: primitive: 232.
:: primitive: 232.
||: primitive: 232.
:= primitive: 232.
; primitive: 232.
< primitive: 955.
<= primitive: 955.
<> primitive: 955.
= primitive: 955.
=:|> primitive: 1185.
|=:> primitive: 1185.
|=:|>> primitive: 1185.
|=:|> primitive: 1185.
=:| primitive: 1185.
|=:| primitive: 1185.
|=: primitive: 1185.
=: primitive: 1185.
=>: 733.
> primitive: 955.
>= primitive: 955.
>>: 918, 1102.
>: 1104.
??: 307, 309, 499, 500.
???: 303, 304, 420, 503.
?: 117, 695.
[primitive: 232.
] primitive: 232.
{ primitive: 232.
\ primitive: 232.
#@! primitive: 753.
@!# primitive: 753.
@! primitive: 753.

} primitive: 232.
__GNU_MP_VERSION: 1069.
__GNU_MP_VERSION_MINOR: 1069.
__GNU_MP_VERSION_PATCHLEVEL: 1069.
__LINE__: 6, 173, 252, 256, 624.
_iob: 1054.
_IONBF: 16.
A: 15, 101, 102, 177, 213, 214, 236, 237, 238, 252, 253, 256, 260.
a: 15, 218, 358, 359, 375, 397, 404, 405, 409, 467, 468, 786, 787, 849, 851, 852, 979, 1027, 1060.
a font metric dimension...: 1223.
A secondary expression...: 944.
A tertiary expression...: 946.
a_aux: 397, 398, 399.
a_goal: 396, 397, 398, 401, 403, 408.
a_new: 397, 398, 399.
a_orig: 405, 467.
$a\_tension$: 364, 365.
a_tot: 409.
aa: 353, 354, 356, 357.
ab: 405, 406.
ab_vs_cd: 15, 158, 372, 437, 441, 444, 446, 456, 515, 566, 568, 569, 574, 575, 600, 601, 993, 997, 1009.
$ab\_vs\_cd\_func$: 15.
ab_vs_cd1: 575.
ab_vs_cd2: 575.
abc: 405.
abs: 15, 94, 456.
abs_a: 358.
abs_du: 562.
abs_dv: 562.
abs_tyy: 1010.
abs_x: 157, 158, 450, 600, 1210.
abs_y: 450, 600.
$absdenom$: 931.
$absdet$: 590.
$absent$: 679, 712, 713, 714, 717, 739, 1066.
$absm$: 1172.
$absnum$: 931.
$absorbing$: 719, 725, 727, 797.
$absp$: 646, 1043.
$absv$: 633, 637, 641.

from_scaled: 15, 456.

from_substraction: 15, 456.

frozen_bad_vardef: 216, 227, 763.

frozen_colon: 216, 232, 820.

frozen_dump: 216, 713, 1071.

frozen_end_def: 216, 725, 745.

frozen_end_for: 216, 725, 745.

frozen_end_group: 216, 232, 725, 761.

frozen_etex: 216, 722, 735.

frozen_fi: 216, 721, 809.

frozen_inaccessible: 216, 227, 231, 755.

frozen_left_bracket: 216, 232, 932, 936.

frozen_mpx_break: 216, 732, 735.

frozen_repeat_loop: 216, 828, 829, 830.

frozen_right_delimiter: 216, 227, 725.

frozen_semicolon: 216, 232, 725.

frozen_slash: 216, 931, 955.

frozen_symbols: 216, 221, 222, 228.

frozen_undefined: 216, 227, 289.

frst: 1251.

ftemplate: 1251.

ftype: 41, 44, 45, 46, 52, 1059, 1060.

FUNCTION_TRACE1:   6, 293, 296, 328, 424.

FUNCTION_TRACE2:   6, 178, 179, 184, 239, 240, 243, 254, 263, 265, 287, 294, 295, 350, 358, 652, 922, 941.

FUNCTION_TRACE3:   6, 176, 179, 212, 213, 236, 237, 242, 252, 260, 289, 624, 650, 651, 653.

FUNCTION_TRACE3X: 6.

FUNCTION_TRACE4:   6, 224, 252, 256, 624, 649.

fwrite: 56, 1060.

g: 1143.

gamma: 364, 365.

gamma_orig: 365.

get_arc_time: 410.

get_dep_info: 624, 625.

get_mp_sym_info: 176, 177.

get_mp_sym_sym: 176, 177.

get_next: 107, 115, 667, 718, 719, 728, 729, 734, 860.

get_symbol: 756, 828.

get_symbolic_node: 178.

get_t_next: 734, 737, 748, 755, 758, 761, 765, 777, 781, 783, 797, 811, 1035, 1050, 1108, 1116.

get_x_next: 765, 769, 780, 817, 820, 821, 903, 904, 936, 939, 944, 1032, 1130.

given: 300, 305, 333, 344, 952.

glyph: 1271.

envelope primitive: 955.

gmp_version: 5, 1069.

gr_black_val: 1270.

gr_color_model: 1270.

gr_cyan_val: 1270.

gr_dash_p: 1270.

gr_depth_val: 1270.

gr_font_dsize: 1270.

gr_font_n: 1270.

gr_font_name: 1270.

gr_height_val: 1270.

gr_htap_p: 1270.

gr_lcap_val: 1270.

gr_link: 1268, 1270.

gr_ljoin_val: 1270.

gr_magenta_val: 1270.

gr_miterlim_val: 1270.

gr_object: 493.

gr_path_p: 1270, 1271.

gr_pen_p: 1270, 1271.

gr_post_script: 1270.

gr_pre_script: 1268, 1270.

gr_text_l: 1270.

gr_text_p: 1270.

gr_tx_val: 1270.

gr_txx_val: 1270.

gr_txy_val: 1270.

gr_ty_val: 1270.

gr_type: 1271.

gr_tyx_val: 1270.

gr_tyy_val: 1270.

gr_width_val: 1270.

gr_XXXX: 1270.

gr_yellow_val: 1270.

greater: 15, 456.

green: 458, 460, 461, 462, 464, 465, 469, 471, 472, 494, 502, 516, 973, 1125.

green_part: 270, 272, 281, 912, 922, 931, 940, 961, 965, 969, 982, 988, 990, 994, 996, 1041, 1125.

*green_part_*:   270.

greenpart primitive: 955.

grey: 458, 502, 973, 1125, 1271.

grey_part: 270, 281.

greypart primitive: 955.

group_line: 931.

gs_colormodel: 201.

h: 302, 303, 310, 311, 328, 393, 409, 410, 415, 416, 418, 419, 422, 423, 424, 433, 434, 446, 453, 483, 484, 485, 486, 487, 489, 496, 497, 505, 506, 510, 526, 527, 532, 533, 544, 580, 600, 619, 943, 1009, 1045, 1059, 1060, 1241, 1269, 1270, 1271, 1272, 1273, 1274, 1275.

half: 15, 456, 613.

half_cos: 429, 430, 431, 432.

half_error_line: 26, 28, 692, 698, 699, 700.

quad: 1166.

quad_code: <u>1166</u>.

quarterword: 159, 160, <u>161</u>, 187, 188, 191, 245, 266, 269, 272, 275, 288, 290, 382, 387, 424, 475, 492, 580, 631, 632, 639, 640, 641, 642, 653, 655, 673, 705, 711, 712, 714, 729, 747, 748, 761, 788, 789, 803, 807, 813, 815, 817, 831, 906, 907, 915, 923, 931, 936, 952, 956, 957, 959, 960, 961, 962, 963, 967, 968, 969, 973, 974, 982, 985, 988, 989, 991, 992, 993, 997, 998, 1005, 1009, 1019, 1030, 1043, 1047, 1049, 1130, 1131, 1132, 1134, 1136, 1171, 1172, 1179, 1180, 1203, 1270.

quote: 748, <u>753</u>.

quote primitive: 753.

quoted_filename: <u>844</u>, 846, 847.

qx: <u>580</u>.

qy: <u>580</u>.

q0: <u>544</u>, 549, <u>580</u>, <u>1125</u>.

q1: 256.

r: <u>15</u>, <u>158</u>, <u>245</u>, <u>250</u>, <u>280</u>, <u>282</u>, <u>283</u>, <u>285</u>, <u>287</u>, <u>327</u>, <u>350</u>, <u>382</u>, <u>434</u>, <u>486</u>, <u>529</u>, <u>544</u>, <u>552</u>, <u>580</u>, <u>588</u>, <u>590</u>, <u>637</u>, <u>638</u>, <u>639</u>, <u>641</u>, <u>642</u>, <u>648</u>, <u>649</u>, <u>653</u>, <u>654</u>, <u>655</u>, <u>657</u>, <u>663</u>, <u>664</u>, <u>758</u>, <u>761</u>, <u>784</u>, <u>923</u>, <u>931</u>, <u>932</u>, <u>942</u>, <u>945</u>, <u>948</u>, <u>957</u>, <u>965</u>, <u>988</u>, <u>990</u>, <u>991</u>, <u>996</u>, <u>998</u>, <u>1009</u>, <u>1019</u>, <u>1021</u>, <u>1023</u>, <u>1042</u>, <u>1043</u>, <u>1179</u>, <u>1180</u>, <u>1195</u>, <u>1201</u>.

r_delim: <u>761</u>, <u>784</u>, 791, 792, 793, 795, <u>796</u>, <u>797</u>, 798, 802, <u>931</u>, <u>1087</u>, <u>1088</u>, <u>1089</u>.

R_OK: 44, 46.

r_packets: <u>605</u>, 613, 617.

rabs: <u>654</u>.

Ramshaw, Lyle Harold: 1158.

random_seed: 150, <u>151</u>, 152, 1066, 1299.

randoms: <u>150</u>, 152, 153, 155, 156.

rd_file: <u>893</u>, 894, 895, <u>985</u>, 1289, 1290.

rd_fname: <u>893</u>, 894, 895, <u>985</u>, 1289, 1290.

read_ascii_file: 16, <u>43</u>, 64, 1061.

read_binary_file: 16, 43, 1061.

read_files: <u>893</u>, 985, 1289.

read_font_info: 1234.

readfrom primitive: 892, 955.

readstring primitive: 955.

readf_index: <u>892</u>, 893, 895, 896, 897, 985.

reading_preload: <u>138</u>, 713, 1284.

realloc: 53, 171.

realmode: <u>46</u>, <u>1060</u>.

recursion: 107, 115, 244, 284, 396, 485, 532, 765, 782, 817, 898, 1036, 1104.

red: <u>458</u>, 460, 461, <u>462</u>, 464, 465, <u>469</u>, 471, 472, 494, 502, 516, 973, 1125.

red_part: <u>270</u>, 272, 281, 912, 922, 931, 940, 961, 965, 969, 982, 988, 990, 994, 996, 1041, 1125.

*red_part_*: <u>270</u>.

redpart primitive: 955.

ref_count: <u>248</u>, 249.

*ref_count_*: <u>482</u>, 484.

reference counts: 248, 684.

refs: <u>72</u>, 749, 987.

relax: 765.

rem_byte: <u>1164</u>, 1184, 1189, 1220.

remainder: 1162, 1163, 1164, 1167.

rep: 1165.

repeat_loop: 765.

report_redundant_or_inconsistent_equation: <u>1041</u>.

res: <u>979</u>, 980.

reset_stream: <u>1060</u>.

RESTART: <u>343</u>, <u>729</u>, <u>755</u>, <u>931</u>, 936, <u>940</u>, <u>944</u>, 946, <u>947</u>, <u>1041</u>.

restart: 729, 732, 936.

restore_cur_exp: <u>907</u>.

resume_iteration: 831, 835.

resume_operation: 821.

RESWITCH: <u>817</u>.

ret: <u>15</u>, <u>156</u>, <u>157</u>, <u>158</u>, <u>354</u>, <u>364</u>, <u>365</u>, <u>396</u>, 397, 401, 403, <u>404</u>, <u>405</u>, <u>408</u>, <u>409</u>, <u>410</u>, 412, 413, <u>434</u>, <u>452</u>, <u>467</u>, <u>468</u>, <u>489</u>, <u>504</u>, <u>524</u>, <u>583</u>, <u>600</u>, <u>641</u>, <u>642</u>, <u>655</u>, <u>923</u>, <u>931</u>, <u>977</u>, <u>978</u>, <u>979</u>, <u>981</u>, <u>988</u>, <u>997</u>, <u>1010</u>, <u>1015</u>, <u>1017</u>, <u>1023</u>, <u>1026</u>, <u>1200</u>, <u>1210</u>.

RETURN: <u>410</u>, 413, <u>561</u>.

retval: <u>376</u>.

ret1: <u>991</u>.

ret2: <u>991</u>.

reverse primitive: 955.

reversed: <u>1027</u>.

right_coord: 387.

right_curl: <u>300</u>, 309, 331, 344, 362, 375, 948.

right_delimiter: 1085.

right_given: <u>300</u>, 309, 344, 361, 373, 948.

right_paren_class: <u>210</u>, 211, 245.

right_tension: <u>300</u>, 304, 306, 354, 355, 362, 363, 371, 372, 374, 375, 948.

right_type: <u>297</u>, <u>298</u>, 317, 320.

right_x: <u>297</u>, <u>298</u>, 300, 307, 313, 316, 317, 320, 324, 327, 331, 334, 344, 371, 374, 375, 378, 382, 387, 389, 409, 410, 414, 416, 417, 421, 425, 427, 429, 451, 467, 515, 528, 549, 552, 554, 559, 579, 585, 588, 589, 596, 597, 598, 601, 617, 620, 948, 980, 1007, 1008, 1030.

right_y: <u>297</u>, <u>298</u>, 300, 307, 313, 316, 317, 320, 324, 327, 331, 334, 344, 371, 374, 375, 378, 382, 387, 389, 409, 410, 414, 416, 417, 421, 425, 427, 429, 451, 467, 528, 549, 552, 554, 559, 579, 585, 588, 589, 596, 597, 598, 601, 617, 620, 948, 980, 1007, 1008, 1030.

⟨ Add offset w to the cubic from p to q 585 ⟩   Used in section 580.

⟨ Adjust θ_n to equal θ_0 and **goto** *found* 357 ⟩   Used in section 353.

⟨ Adjust the balance for a delimited argument; **break** if done 798 ⟩   Used in section 797.

⟨ Adjust the balance for an undelimited argument; **break** if done 799 ⟩   Used in section 797.

⟨ Adjust the balance; **break** if it's zero 752 ⟩   Used in section 748.

⟨ Adjust $bbmin[c]$ and $bbmax[c]$ to accommodate x 388 ⟩   Used in sections 387, 391, and 392.

⟨ Adjust p's bounding box to contain $str\_pool[k]$; advance k 1242 ⟩   Used in section 1241.

⟨ Advance dd until finding the first dash that overlaps dln when offset by $xoff$ 523 ⟩   Used in section 522.

⟨ Advance p making sure the links are OK and **return** if there is a problem 420 ⟩   Used in section 419.

⟨ Advance p to node q, removing any "dead" cubics that might have been introduced by the splitting process 549 ⟩   Used in section 544.

⟨ Advance p to the end of the path and make q the previous knot 531 ⟩   Used in section 527.

⟨ Allocate or initialize variables 28, 49, 51, 61, 74, 79, 100, 110, 120, 139, 143, 152, 164, 167, 196, 221, 606, 675, 682, 686, 867, 888, 894, 1168, 1228, 1280 ⟩   Used in section 16.

⟨ Append the current expression to arg_list 794 ⟩   Used in sections 792 and 800.

⟨ Back up an outer symbolic token so that it can be reread 723 ⟩   Used in section 721.

⟨ Basic printing procedures 85, 86, 87, 88, 89, 91, 92, 147, 188, 207, 209, 849 ⟩   Used in section 5.

⟨ Begin the progress report for the output of picture c 1258 ⟩   Used in section 1273.

⟨ Bisect the Bézier quadratic given by $dx0$, $dy0$, $dx1$, $dy1$, $dx2$, $dy2$ 400 ⟩   Used in section 396.

⟨ Calculate the given value of θ_n and **goto** *found* 360 ⟩   Used in section 350.

⟨ Calculate the ratio $ff = C_k/(C_k + B_k - u_{k-1}A_k)$ 355 ⟩   Used in section 353.

⟨ Calculate the turning angles ψ_k and the distances $d_{k,k+1}$; set n to the length of the path 343 ⟩   Used in section 339.

⟨ Calculate the values $aa = A_k/B_k$, $bb = D_k/C_k$, $dd = (3 - \alpha_{k-1})d_{k,k+1}$, $ee = (3 - \beta_{k+1})d_{k-1,k}$, and $cc = (B_k - u_{k-1}A_k)/B_k$ 354 ⟩   Used in section 353.

⟨ Calculate the values of v_k and w_k 356 ⟩   Used in section 353.

⟨ Cases for printing graphical object node p 498, 503, 507, 508, 509 ⟩   Used in section 497.

⟨ Cases of $print\_cmd\_mod$ for symbolic printing of primitives 233, 736, 746, 754, 760, 772, 810, 956, 1047, 1072, 1079, 1082, 1100, 1106, 1123, 1129, 1144, 1176, 1186 ⟩   Used in section 671.

⟨ Change one-point paths into dead cycles 620 ⟩   Used in section 619.

⟨ Change the interaction level and **return** 125 ⟩   Used in section 123.

⟨ Character k cannot be printed 78 ⟩   Used in section 87.

⟨ Check for retracing between knots qq and rr and **goto** *not_found* if there is a problem 515 ⟩   Used in section 514.

⟨ Check for the assignment in a loop header 826 ⟩   Used in section 825.

⟨ Check for the presence of a colon 827 ⟩   Used in section 825.

⟨ Check if the file has ended while flushing TeX material and set the result value for $check\_outer\_validity$ 722 ⟩   Used in section 721.

⟨ Check rotation direction based on node position 575 ⟩   Used in section 574.

⟨ Check that the proper right delimiter was present 793 ⟩   Used in section 792.

⟨ Check the "constant" values for consistency 30, 608 ⟩   Used in section 16.

⟨ Check the control points against the bounding box and set $wavy\colon = true$ if any of them lie outside 389 ⟩   Used in section 387.

⟨ Clip the bounding box in h to the rectangle given by $x0a$, $x1a$, $y0a$, $y1a$ 541 ⟩   Used in section 539.

⟨ Close all open files in the rd_file and wr_file arrays 1289 ⟩   Used in section 1287.

⟨ Compain that the edge structure contains a node of the wrong type and **goto** *not_found* 511 ⟩   Used in section 510.

⟨ Compile a ligature/kern command 1189 ⟩   Used in section 1184.

⟨ Complain about a character tag conflict 1181 ⟩   Used in section 1180.

⟨ Complain about a misplaced **etex** 744 ⟩   Used in section 739.

⟨ Complain about a misplaced **mpxbreak** 743 ⟩   Used in section 739.

⟨ Complain about improper map operation 1246 ⟩   Used in section 1245.

⟨ Complain about improper special operation 1266 ⟩   Used in section 1265.

⟨ Complain that MPX files cannot contain TEX material 741 ⟩   Used in section 739.

⟨ Complain that it's not a known picture 1139 ⟩   Used in section 1138.

⟨ Complain that we are not reading a file 742 ⟩   Used in section 739.

⟨ Complete the error message, and set *cur_sym* to a token that might help recover from the error 725 ⟩   Used in section 724.

⟨ Complete the offset splitting process 570 ⟩   Used in section 555.

⟨ Compute a check sum in $(b1, b2, b3, b4)$ 1215 ⟩   Used in section 1214.

⟨ Compute test coefficients $(t0, t1, t2)$ for $d(t)$ versus d_k or d_{k-1} 562 ⟩   Used in sections 561 and 570.

⟨ Compute the ligature/kern program offset and implant the left boundary label 1220 ⟩   Used in section 1218.

⟨ Constants in the outer block 23 ⟩   Used in section 4.

⟨ Copy the bounding box information from *h* to *hh* and make *bblast*(*hh*) point into the new object list 490 ⟩   Used in section 487.

⟨ Copy the dash list from *h* to *hh* 488 ⟩   Used in section 487.

⟨ Deal with a negative *arc0_orig* value and **return** 412 ⟩   Used in section 410.

⟨ Dealloc variables 27, 62, 75, 80, 153, 168, 222, 341, 346, 369, 386, 432, 449, 607, 612, 616, 676, 683, 688, 843, 855, 869, 876, 928, 1064, 1098, 1169, 1199, 1213, 1229, 1257, 1281, 1290 ⟩   Used in section 12.

⟨ Decide on the net change in pen offsets and set *turn_amt* 574 ⟩   Used in section 555.

⟨ Declarations 8, 45, 70, 84, 95, 101, 107, 121, 177, 187, 205, 206, 214, 217, 223, 238, 241, 244, 246, 253, 255, 264, 279, 284, 286, 302, 310, 312, 314, 326, 347, 349, 359, 364, 370, 404, 418, 422, 433, 439, 468, 485, 491, 496, 501, 505, 512, 533, 551, 553, 556, 560, 567, 587, 622, 625, 627, 631, 636, 640, 643, 645, 647, 661, 666, 670, 680, 689, 708, 710, 726, 728, 731, 738, 750, 765, 780, 783, 786, 788, 796, 845, 856, 889, 896, 906, 917, 921, 924, 950, 954, 967, 972, 1033, 1036, 1038, 1042, 1044, 1056, 1059, 1088, 1095, 1171, 1234, 1238, 1240, 1272, 1274, 1283, 1288, 1296 ⟩   Used in section 5.

⟨ Declare a procedure called *no_string_err* 1148 ⟩   Used in section 1145.

⟨ Declare action procedures for use by *do_statement* 1048, 1074, 1083, 1086, 1091, 1093, 1101, 1103, 1107, 1109, 1111, 1115, 1117, 1119, 1124, 1126, 1131, 1133, 1135, 1137, 1145, 1153, 1177, 1179, 1182, 1244, 1264 ⟩   Used in section 1033.

⟨ Declare binary action procedures 989, 990, 991, 993, 996, 997, 998, 1005, 1006, 1007, 1008, 1009, 1019, 1027, 1028, 1029, 1030, 1031 ⟩   Used in section 988.

⟨ Declare helpers 165 ⟩   Used in section 4.

⟨ Declare miscellaneous procedures that were declared *forward* 247 ⟩   Used in section 1285.

⟨ Declare nullary action procedure 958 ⟩   Used in section 957.

⟨ Declare subroutines for parsing file names 861, 863 ⟩   Used in section 10.

⟨ Declare subroutines needed by *big_trans* 1021, 1023, 1024, 1026 ⟩   Used in section 1019.

⟨ Declare subroutines needed by *make_exp_copy* 941, 942 ⟩   Used in section 940.

⟨ Declare the PostScript output procedures 1269 ⟩   Used in section 1137.

⟨ Declare the basic parsing subroutines 931, 932, 943, 944, 946, 947, 948, 953 ⟩   Used in section 1285.

⟨ Declare the procedure called *dep_finish* 992 ⟩   Used in section 991.

⟨ Declare the procedure called *make_eq* 1040 ⟩   Used in section 1036.

⟨ Declare the procedure called *make_exp_copy* 940 ⟩   Used in section 707.

⟨ Declare the procedure called *print_dp* 915 ⟩   Used in section 906.

⟨ Declare the stashing/unstashing routines 903, 904 ⟩   Used in section 906.

⟨ Declare unary action procedures 960, 961, 962, 963, 964, 965, 966, 969, 973, 974, 975, 976, 977, 978, 980, 981, 982, 983, 984, 985 ⟩   Used in section 959.

⟨ Decrease the string reference count, if the current token is a string 812 ⟩   Used in sections 127, 811, and 1050.

⟨ Decrease the velocities, if necessary, to stay inside the bounding triangle 372 ⟩   Used in section 371.

⟨ Define an extensible recipe 1190 ⟩   Used in section 1183.

⟨ Delete tokens and **continue** 127 ⟩   Used in section 123.

⟨ Descend the structure 1113 ⟩   Used in section 1112.

⟨ Determine the number *n* of arguments already supplied, and set *tail* to the tail of *arg_list* 790 ⟩   Used in section 784.

⟨ Display a cmykcolor node 913 ⟩   Used in section 908.

⟨ Display a color node 912 ⟩   Used in section 908.

⟨ Display a complex type 914 ⟩   Used in section 908.
⟨ Display a pair node 910 ⟩   Used in section 908.
⟨ Display a transform node 911 ⟩   Used in section 908.
⟨ Display a variable macro 1114 ⟩   Used in section 1112.
⟨ Display a variable that's been declared but not defined 916 ⟩   Used in section 908.
⟨ Display big node item v 909 ⟩   Used in sections 910, 911, 912, and 913.
⟨ Display the boolean value of cur_exp 819 ⟩   Used in section 817.
⟨ Display the current context 693 ⟩   Used in section 692.
⟨ Do a Gramm scan and remove vertices where there is no left turn 444 ⟩   Used in section 434.
⟨ Do all the finishing work on the TFM file 1291 ⟩   Used in section 1287.
⟨ End progress report 1259 ⟩   Used in section 1273.
⟨ Enumeration types 185, 186, 189 ⟩   Used in section 4.
⟨ Error handling procedures 112, 114, 132, 135, 137 ⟩   Used in section 5.
⟨ Estimate when the arc length reaches a_goal and set arc_test to that time minus two 403 ⟩   Used in section 396.
⟨ Exit a loop if the proper time has come 775 ⟩   Used in section 769.
⟨ Exit prematurely from an iteration 776 ⟩   Used in section 775.
⟨ Expand the token after the next token 777 ⟩   Used in section 769.
⟨ Explain that the MPX file can't be read and $succumb$ 891 ⟩   Used in section 884.
⟨ Explain what output files were written 1260 ⟩   Used in section 1287.
⟨ Export pending specials 1268 ⟩   Used in section 1270.
⟨ Exported function headers 18, 116, 133, 197, 377, 379, 1053, 1062, 1070, 1237, 1293 ⟩   Used in section 3.
⟨ Exported types 15, 42, 72, 98, 104, 118, 162, 297, 298, 301, 886, 1054, 1276 ⟩   Used in section 3.
⟨ Extract the transformation parameters from the elliptical pen h 427 ⟩   Used in section 426.
⟨ Feed the arguments and replacement text to the scanner 803 ⟩   Used in section 784.
⟨ Fill in the control information between consecutive breakpoints p and q 339 ⟩   Used in section 333.
⟨ Fill in the control points between p and the next breakpoint, then advance p to that breakpoint 333 ⟩   Used in section 328.
⟨ Find and load preload file, if required 854 ⟩   Used in section 16.
⟨ Find any knots on the path from l to r above the l-r line and move them past r 437 ⟩   Used in section 434.
⟨ Find any knots on the path from s to l below the l-r line and move them past l 441 ⟩   Used in section 434.
⟨ Find the bounding box of an elliptical pen 454 ⟩   Used in section 453.
⟨ Find the final direction $(dxin, dyin)$ 565 ⟩   Used in section 555.
⟨ Find the first breakpoint, h, on the path; insert an artificial breakpoint if the path is an unbroken cycle 332 ⟩   Used in section 328.
⟨ Find the first t where $d(t)$ crosses d_{k-1} or set $t := fraction\_one + 1$ 572 ⟩   Used in section 570.
⟨ Find the initial direction (dx, dy) 564 ⟩   Used in section 555.
⟨ Find the minimum lk_offset and adjust all remainders 1221 ⟩   Used in section 1220.
⟨ Find the non-constant part of the transformation for h 451 ⟩   Used in section 450.
⟨ Find the offset for (x, y) on the elliptical pen h 450 ⟩   Used in section 446.
⟨ Find n where $wr\_fname[n] = cur\_exp$ and call $open\_write\_file$ if cur_exp must be inserted 1156 ⟩   Used in section 1155.
⟨ Finish choosing angles and assigning control points 366 ⟩   Used in section 350.
⟨ Finish non-interactive use 1065 ⟩   Used in section 12.
⟨ Finish printing the dash pattern that p refers to 504 ⟩   Used in section 503.
⟨ Finish the TFM file 1217 ⟩   Used in section 1291.
⟨ Fix anything in graphical object pp that should differ from the corresponding field in p 494 ⟩   Used in section 493.
⟨ Fix the offset change in $mp\_knot\_info(c)$ and set c to the return value of $offset\_prep$ 569 ⟩   Used in section 544.
⟨ Fix up $mp{\to}internal[mp\_job\_name]$ 868 ⟩   Used in sections 16, 875, 880, 1066, and 1251.
⟨ Flush the TEX material 740 ⟩   Used in section 739.
⟨ Flush the dash list, recycle h and return Λ 519 ⟩   Used in section 510.

⟨ Flush *name* and replace it with *cur_name* if it won't be needed 881 ⟩   Used in section 880.

⟨ For each of the eight cases, change the relevant fields of *cur_exp* and **goto** *done*; but do nothing if capsule *p* doesn't have the appropriate type 1002 ⟩   Used in section 998.

⟨ Free table entries 183, 259, 480, 629, 669, 764, 901, 971, 1001, 1194, 1208 ⟩   Used in section 12.

⟨ Get the first line of input and prepare to start 1298 ⟩   Used in section 16.

⟨ Get the linear equations started; or **return** with the control points in place, if linear equations needn't be solved 351 ⟩   Used in section 350.

⟨ Get user's advice and **return** 117 ⟩   Used in section 115.

⟨ Give reasonable values for the unused control points between *p* and *q* 334 ⟩   Used in section 333.

⟨ Global variables 14, 25, 29, 37, 47, 60, 65, 73, 76, 77, 105, 109, 111, 138, 142, 150, 166, 175, 181, 194, 208, 210, 216, 225, 291, 325, 340, 345, 367, 384, 430, 447, 543, 545, 604, 605, 610, 614, 623, 634, 667, 674, 679, 685, 691, 719, 730, 762, 766, 807, 822, 841, 844, 865, 893, 899, 926, 929, 986, 999, 1057, 1130, 1141, 1150, 1158, 1167, 1197, 1205, 1211, 1225, 1227, 1247, 1255, 1263, 1279, 1282 ⟩   Used in section 4.

⟨ Graphical object codes 459, 463, 470, 474, 1267 ⟩   Used in section 457.

⟨ If consecutive knots are equal, join them explicitly 331 ⟩   Used in section 328.

⟨ If endpoint, double the path *c*, and set *spec_p1* and *spec_p2* 595 ⟩   Used in section 580.

⟨ If *dd* has 'fallen off the end', back up to the beginning and fix *xoff* 524 ⟩   Used in section 522.

⟨ If *miterlim* is less than the secant of half the angle at *q* then set *join_type* := 2 583 ⟩   Used in section 582.

⟨ Initializations after first line is read 17 ⟩   Used in section 16.

⟨ Initializations following first line 1299 ⟩   Used in section 1298.

⟨ Initialize for intersections at level zero 617 ⟩   Used in section 613.

⟨ Initialize table entries 182, 202, 203, 226, 227, 258, 368, 385, 448, 479, 611, 615, 628, 668, 763, 830, 927, 970, 1000, 1193, 1198, 1207, 1212 ⟩   Used in section 1297.

⟨ Initialize the incoming direction and pen offset at *c* 548 ⟩   Used in section 544.

⟨ Initialize the input routines 717, 720 ⟩   Used in section 1298.

⟨ Initialize the output routines 81, 90 ⟩   Used in sections 16 and 1066.

⟨ Initialize the pen size *n* 547 ⟩   Used in section 544.

⟨ Initialize the random seed to *cur_exp* 1076 ⟩   Used in section 1075.

⟨ Initialize *p* as the *k*th knot of a circle of unit diameter, transforming it appropriately 429 ⟩   Used in section 426.

⟨ Initialize *v002*, *v022*, and the arc length estimate *arc*; if it overflows set *arc_test* and **return** 401 ⟩   Used in section 396.

⟨ Initiate or terminate input from a file 773 ⟩   Used in section 769.

⟨ Insert a dash between *d* and *dln* for the overlap with the offset version of *dd* 525 ⟩   Used in section 522.

⟨ Insert a new knot *r* between *p* and *q* as required for a mitered join 590 ⟩   Used in section 589.

⟨ Insert *d* into the dash list and **goto** *not_found* if there is an error 517 ⟩   Used in section 510.

⟨ Install a complex multiplier, then **goto** *done* 1004 ⟩   Used in section 1002.

⟨ Install sines and cosines, then **goto** *done* 1003 ⟩   Used in section 1002.

⟨ Internal library declarations 10, 83, 93, 108, 113, 134, 136, 154, 172, 180, 329, 852, 870, 872, 1096, 1231, 1250, 1253, 1261 ⟩   Used in section 4.

⟨ Interpret code *c* and **return** if done 123 ⟩   Used in section 117.

⟨ Introduce new material from the terminal and **return** 126 ⟩   Used in section 123.

⟨ Local variables for formatting calculations 698 ⟩   Used in section 692.

⟨ Local variables for initialization 35, 149 ⟩   Used in section 13.

⟨ Log the subfile sizes of the TFM file 1224 ⟩   Used in section 1217.

⟨ MPlib header stuff 201, 299, 457 ⟩   Used in section 3.

⟨ MPlib internal header stuff 6, 36, 67, 82, 174, 193, 235, 251, 262, 267, 270, 273, 455, 458, 462, 469, 473, 477, 482, 805 ⟩   Used in section 4.

⟨ Make sure the current expression is a known picture 839 ⟩   Used in section 838.

⟨ Make sure *h* isn't confused with an elliptical pen 417 ⟩   Used in section 415.

⟨ Make sure *p* and *p0* are the same color and **goto** *not_found* if there is an error 516 ⟩   Used in section 514.

⟨ Make the bounding box of *h* unknown if it can't be updated properly without scanning the whole structure 1013 ⟩   Used in section 1009.

⟨ Make the elliptical pen *h* into a path 426 ⟩   Used in section 424.

⟨ Make (dx, dy) the final direction for the path segment from *q* to *p*; set *d* 528 ⟩   Used in section 527.

⟨ Make (xx, yy) the offset on the untransformed **pencircle** for the untransformed version of (x, y) 452 ⟩   Used in section 450.

⟨ Make *c* look like a cycle of length one 596 ⟩   Used in section 595.

⟨ Make *d* point to a new dash node created from stroke *p* and path *pp* or **goto** *not_found* if there is an error 514 ⟩   Used in section 510.

⟨ Make *mp_link*(*pp*) point to a copy of object *p*, and update *p* and *pp* 493 ⟩   Used in section 492.

⟨ Make *q* a capsule containing the next picture component from *loop_list*(*loop_ptr*) or **goto** *not_found* 834 ⟩   Used in section 831.

⟨ Make *r* the last of two knots inserted between *p* and *q* to form a squared join 591 ⟩   Used in section 589.

⟨ Make *ss* negative if and only if the total change in direction is more than 180° 576 ⟩   Used in section 574.

⟨ Massage the TFM heights, depths, and italic corrections 1206 ⟩   Used in section 1291.

⟨ Massage the TFM widths 1204 ⟩   Used in section 1291.

⟨ Metapost version header 2 ⟩   Used in section 3.

⟨ Normalize the direction (dx, dy) and find the pen offset (xx, yy) 529 ⟩   Used in section 527.

⟨ Operation codes 190 ⟩   Used in section 189.

⟨ Option variables 26, 43, 48, 50, 66, 99, 119, 151, 163, 195, 853, 866, 887, 1277 ⟩   Used in sections 3 and 4.

⟨ Other cases for updating the bounding box based on the type of object *p* 534, 535, 537, 538, 539 ⟩   Used in section 532.

⟨ Other local variables for *make_choices* 342 ⟩   Used in section 328.

⟨ Other local variables for *make_envelope* 584, 592 ⟩   Used in section 580.

⟨ Other local variables for *offset_prep* 558, 573 ⟩   Used in section 544.

⟨ Other local variables in *make_dashes* 521 ⟩   Used in section 510.

⟨ Other local variables in *make_path* 428 ⟩   Used in section 424.

⟨ Output statistics about this job 1292 ⟩   Used in section 1287.

⟨ Output the character information bytes, then output the dimensions themselves 1219 ⟩   Used in section 1217.

⟨ Output the extensible character recipes and the font metric parameters 1223 ⟩   Used in section 1217.

⟨ Output the ligature/kern program 1222 ⟩   Used in section 1217.

⟨ Output the subfile sizes and header bytes 1218 ⟩   Used in section 1217.

⟨ Pop the condition stack 814 ⟩   Used in sections 817, 818, and 820.

⟨ Prepare for derivative computations; **goto** *not_found* if the current cubic is dead 559 ⟩   Used in section 555.

⟨ Prepare for step-until construction and **break** 837 ⟩   Used in section 836.

⟨ Prepare function pointers for non-interactive use 1061 ⟩   Used in section 16.

⟨ Pretend we're reading a new one-line file 779 ⟩   Used in section 778.

⟨ Print an abbreviated value of *v* or *vv* with format depending on *t* 908 ⟩   Used in section 907.

⟨ Print control points between *p* and *q*, then **goto** *done1* 307 ⟩   Used in section 304.

⟨ Print information for a curve that begins *curl* or *given* 309 ⟩   Used in section 304.

⟨ Print information for a curve that begins *open* 308 ⟩   Used in section 304.

⟨ Print information for adjacent knots *p* and *q* 304 ⟩   Used in section 303.

⟨ Print join and cap types for stroked node *p* 500 ⟩   Used in section 503.

⟨ Print join type for graphical object *p* 499 ⟩   Used in sections 498 and 500.

⟨ Print location of current line 694 ⟩   Used in section 693.

⟨ Print string *cur_exp* as an error message 1152 ⟩   Used in section 1146.

⟨ Print tension between *p* and *q* 306 ⟩   Used in section 304.

⟨ Print the banner line, including the date and time 878 ⟩   Used in section 875.

⟨ Print the cubic between *p* and *q* 579 ⟩   Used in section 577.

⟨ Print the current loop value 696 ⟩   Used in section 695.

⟨ Print the elliptical pen *h* 421 ⟩   Used in section 419.

⟨ Print the help information and **continue** 128 ⟩   Used in section 123.

⟨Print the menu of available options 124⟩   Used in section 123.

⟨Print the name of a **vardef**'d macro 697⟩   Used in section 695.

⟨Print the string *err_help*, possibly on several lines 129⟩   Used in sections 128 and 130.

⟨Print two dots, followed by *given* or *curl* if present 305⟩   Used in section 303.

⟨Print two lines using the tricky pseudoprinted information 700⟩   Used in section 693.

⟨Print type of token list 695⟩   Used in section 693.

⟨Process a *skip_to* command and **goto** *done* 1187⟩   Used in section 1184.

⟨Pseudoprint the line 701⟩   Used in section 693.

⟨Pseudoprint the token list 702⟩   Used in section 693.

⟨Push the condition stack 813⟩   Used in section 817.

⟨Put a string into the input buffer 778⟩   Used in section 769.

⟨Put each of METAPOST's primitives into the hash table 200, 232, 735, 745, 753, 759, 771, 809, 955, 1046, 1071, 1078, 1081, 1099, 1122, 1128, 1143, 1175, 1185⟩   Used in section 1297.

⟨Put help message on the transcript file 130⟩   Used in section 115.

⟨Put the desired file name in (*cur_name*, *cur_ext*, *cur_area*) 883⟩   Used in section 880.

⟨Read the first line of the new file 882⟩   Used in sections 880 and 884.

⟨Record a label in a lig/kern subprogram and **goto continue** 1188⟩   Used in section 1184.

⟨Record the end of file on *wr_file*[*n*] 1157⟩   Used in section 1155.

⟨Recycle an independent variable 923⟩   Used in section 922.

⟨Reduce to simple case of straight line and **return** 374⟩   Used in section 351.

⟨Reduce to simple case of two givens and **return** 373⟩   Used in section 351.

⟨Reinitialize the bounding box in header *h* and call *set_bbox* recursively starting at *mp_link*(*p*) 540⟩   Used in section 539.

⟨Remove knot *p* and back up *p* and *q* but don't go past *l* 445⟩   Used in section 444.

⟨Remove the cubic following *p* and update the data structures to merge *r* into *p* 550⟩   Used in section 549.

⟨Remove *open* types at the breakpoints 344⟩   Used in section 339.

⟨Repeat a loop 774⟩   Used in section 769.

⟨Replace an interval of values by its midpoint 1202⟩   Used in section 1201.

⟨Replace *mp_link*(*d*) by a dashed version as determined by edge header *hh* and scale factor *ds* 522⟩   Used in section 520.

⟨Report an unexpected problem during the choice-making 330⟩   Used in section 328.

⟨Rescale if necessary to make sure *a*, *b*, and *c* are all less than EL_GORDO *div* 3 407⟩   Used in section 405.

⟨Reverse the dash list of *h* 1011⟩   Used in section 1010.

⟨Rotate the cubic between *p* and *q*; then **goto** *found* if the rotated cubic travels due east at some time *tt*; but **break** if an entire cyclic path has been traversed 601⟩   Used in section 600.

⟨Save string *cur_exp* as the *err_help* 1149⟩   Used in section 1146.

⟨Save the filename template 1147⟩   Used in section 1146.

⟨Scale the bounding box by *txx* + *txy* and *tyx* + *tyy*; then shift by (*tx*, *ty*) 1015⟩   Used in section 1013.

⟨Scale the dash list by *txx* and shift it by *tx* 1012⟩   Used in section 1010.

⟨Scale up *del1*, *del2*, and *del3* for greater accuracy; also set *del* to the first nonzero element of (*del1*, *del2*, *del3*) 390⟩   Used in section 387.

⟨Scan a suffix with optional delimiters 802⟩   Used in section 800.

⟨Scan a variable primary; **goto** *restart* if it turns out to be a macro 936⟩   Used in section 931.

⟨Scan an expression followed by '**of** ⟨primary⟩' 801⟩   Used in section 800.

⟨Scan file name in the buffer 874⟩   Used in section 873.

⟨Scan the argument represented by *mp_sym_info*(*r*) 795⟩   Used in section 792.

⟨Scan the delimited argument represented by *mp_sym_info*(*r*) 792⟩   Used in section 791.

⟨Scan the loop text and put it on the loop control stack 829⟩   Used in section 825.

⟨Scan the pen polygon between *w0* and *w* and make *max_ht* the range dot product with (*ht_x*, *ht_y*) 593⟩   Used in section 591.

⟨Scan the remaining arguments, if any; set *r* to the first token of the replacement text 791⟩   Used in section 784.

⟨ Scan the values to be used in the loop 836 ⟩   Used in section 825.

⟨ Scan to the matching **mp_stop_bounds_node** node and update p and $bblast(h)$ 536 ⟩   Used in section 535.

⟨ Scan undelimited argument(s) 800 ⟩   Used in section 791.

⟨ Scan $dash\_list(h)$ and deal with any dashes that are themselves dashed 520 ⟩   Used in section 510.

⟨ Scold the user for having an extra **endfor** 770 ⟩   Used in section 769.

⟨ Set initial values of key variables 38, 39, 199, 211, 292, 431, 546, 635, 767, 808, 823, 842, 900, 930, 987, 1142, 1151, 1170, 1232, 1248, 1256 ⟩   Used in section 13.

⟨ Set the height and depth to zero if the bounding box is empty 1243 ⟩   Used in section 1241.

⟨ Set the incoming and outgoing directions at q; in case of degeneracy set $join\_type := 2$ 597 ⟩   Used in section 582.

⟨ Set the outgoing direction at q 598 ⟩   Used in section 597.

⟨ Set up a picture iteration 838 ⟩   Used in section 825.

⟨ Set up equation for a curl at θ_n and **goto** $found$ 363 ⟩   Used in section 350.

⟨ Set up equation to match mock curvatures at z_k; then **goto** $found$ with θ_n adjusted to equal θ_0, if a cycle has ended 353 ⟩   Used in section 350.

⟨ Set up the equation for a curl at θ_0 362 ⟩   Used in section 351.

⟨ Set up the equation for a given value of θ_0 361 ⟩   Used in section 351.

⟨ Set a_new and a_aux so their sum is $2 * a\_goal$ and a_new is as large as possible 398 ⟩   Used in section 397.

⟨ Set $dash\_y(h)$ and merge the first and last dashes if necessary 518 ⟩   Used in section 510.

⟨ Set $join\_type$ to indicate how to handle offset changes at q 582 ⟩   Used in section 580.

⟨ Set l to the leftmost knot in polygon h 435 ⟩   Used in section 434.

⟨ Set $p = mp\_link(p)$ and add knots between p and q as required by $join\_type$ 589 ⟩   Used in section 580.

⟨ Set r to the rightmost knot in polygon h 436 ⟩   Used in section 434.

⟨ Show a numeric or string or capsule token 1105 ⟩   Used in section 1104.

⟨ Show the text of the macro being expanded, and the existing arguments 785 ⟩   Used in section 784.

⟨ Skip to **elseif** or **else** or **fi**, then **goto** $done$ 818 ⟩   Used in section 817.

⟨ Sort the path from l to r by increasing x 442 ⟩   Used in section 434.

⟨ Sort the path from r to l by decreasing x 443 ⟩   Used in section 434.

⟨ Split off another rising cubic for fin_offset_prep 571 ⟩   Used in section 570.

⟨ Split the cubic at t, and split off another cubic if the derivative crosses back 563 ⟩   Used in section 561.

⟨ Split the cubic between p and q, if necessary, into cubics associated with single offsets, after which q should point to the end of the final such cubic 555 ⟩   Used in section 544.

⟨ Start non-interactive work 1066 ⟩   Used in section 1067.

⟨ Step ww and move kk one step closer to $k0$ 594 ⟩   Used in section 593.

⟨ Step w and move k one step closer to $zero\_off$ 586 ⟩   Used in section 580.

⟨ Store a list of font dimensions 1192 ⟩   Used in section 1183.

⟨ Store a list of header bytes 1191 ⟩   Used in section 1183.

⟨ Store a list of ligature/kern steps 1184 ⟩   Used in section 1183.

⟨ Store the width information for character code c 1173 ⟩   Used in section 1138.

⟨ Subdivide for a new level of intersection 618 ⟩   Used in section 613.

⟨ Subdivide the Bézier quadratic defined by a, b, c 406 ⟩   Used in section 405.

⟨ Substitute for cur_sym, if it's on the $subst\_list$ 751 ⟩   Used in section 748.

⟨ Swap the x and y parameters in the bounding box of h 1014 ⟩   Used in section 1013.

⟨ Tell the user what has run away and try to recover 724 ⟩   Used in section 721.

⟨ Terminate the current conditional and skip to **fi** 820 ⟩   Used in section 769.

⟨ Test if the control points are confined to one quadrant or rotating them 45° would put them in one quadrant. Then set $simple$ appropriately 402 ⟩   Used in section 396.

⟨ Test the extremes of the cubic against the bounding box 391 ⟩   Used in section 387.

⟨ Test the second extreme against the bounding box 392 ⟩   Used in section 391.

⟨ The arithmetic progression has ended 832 ⟩   Used in section 831.

⟨ Trace the fraction multiplication 995 ⟩   Used in section 994.

⟨ Trace the start of a loop 833 ⟩   Used in section 831.

⟨ Transform a known big node 1022 ⟩   Used in section 1019.

⟨ Transform an unknown big node and **return** 1020 ⟩   Used in section 1019.

⟨ Transform graphical object q 1016 ⟩   Used in section 1009.

⟨ Transform known by known 1025 ⟩   Used in section 1022.

⟨ Transform the compact transformation 1018 ⟩   Used in section 1016.

⟨ Transform $mp\_pen\_p(qq)$, making sure polygonal pens stay counter-clockwise 1017 ⟩   Used in section 1016.

⟨ Try to get a different log file name 877 ⟩   Used in section 875.

⟨ Try to transform the dash list of h 1010 ⟩   Used in section 1009.

⟨ Types in the outer block 33, 34, 41, 161, 192, 215, 248, 290, 383, 478, 673, 747, 821, 892, 1058, 1226 ⟩   Used in section 4.

⟨ Update a_new to reduce $a\_new + a\_aux$ by a 399 ⟩   Used in section 397.

⟨ Update arc and t_tot after do_arc_test has just returned t 411 ⟩   Used in section 410.

⟨ Update $mp\_knot\_info(p)$ and find the offset w_k such that $d_{k-1} \preceq (dx, dy) \prec d_k$; also advance $w0$ for the direction change at p 566 ⟩   Used in section 555.

⟨ Update t_tot and arc to avoid going around the cyclic path too many times but set $arith\_error := true$ and **goto** $done$ on overflow 413 ⟩   Used in section 410.

⟨ Update w as indicated by $mp\_knot\_info(p)$ and print an explanation 578 ⟩   Used in section 577.

⟨ Use one or two recursive calls to compute the arc_test function 397 ⟩   Used in section 396.

⟨ Use (dx, dy) to generate a vertex of the square end cap and update the bounding box to accommodate it 530 ⟩   Used in section 527.

⟨ Use c to compute the file extension s 1252 ⟩   Used in section 1251.

⟨ Use $offset\_prep$ to compute the envelope spec then walk h around to the initial offset 581 ⟩   Used in section 580.

⟨ Write t to the file named by cur_exp 1155 ⟩   Used in section 1154.

⟨ copy the coordinates of knot p into its control points 425 ⟩   Used in section 424.

⟨ `mplib.h`   3 ⟩

⟨ `mpmp.h`   4 ⟩

MetaPost