The MFT processor

(Version 2.1, January 2021)

The preparation of this report was supported in part by the National Science Foundation under grants IST-8201926, MCS-8300984, and CCR-8610181, and by the System Development Foundation.

1* Introduction. This program converts a METAFONT or METAPOST source file to a TEX file. It was written by D. E. Knuth in June, 1985; a somewhat similar SAIL program had been developed in January, 1980. Changes for METAPOST by W. Bzyl in July, 2001.

The general idea is to input a file called, say, `foo.mf` and to produce an output file called, say, `foo.tex`. The latter file, when processed by TEX, will yield a "prettyprinted" representation of the input file.

Line breaks in the input are carried over into the output; moreover, blank spaces at the beginning of a line are converted to quads of indentation in the output. Thus, the user has full control over the indentation and line breaks. Each line of input is translated independently of the others.

A slight change to METAFONT's comment convention allows further control. Namely, '`%%`' indicates that the remainder of an input line should be copied verbatim to the output; this interrupts the translation and forces `MFT` to produce a certain result.

Furthermore, '`%%%` ⟨ token$_1$ ⟩ ... ⟨ token$_n$ ⟩' introduces a change in `MFT`'s formatting rules; all tokens after the first will henceforth be translated according to the current conventions for ⟨ token$_1$ ⟩. The tokens must be symbolic (i.e., not numeric or string tokens). For example, the input line

```
%%% addto fill draw filldraw
```

says that the '`fill`', '`draw`', and '`filldraw`' operations of plain METAFONT should be formatted as the primitive token '`addto`', i.e., in boldface type. (Without such reformatting commands, `MFT` would treat '`fill`' like an ordinary tag or variable name. In fact, you need a reformatting command even to get parentheses to act like delimiters!)

METAFONT comments, which follow a single `%` sign, should be valid TEX input. But METAFONT material can be included in `|...|` within a comment; this will be translated by `MFT` as if it were not in a comment. For example, a phrase like '`make |x2r| zero`' will be translated into '`make $x_{2r}$ zero`'.

The rules just stated apply to lines that contain one, two, or three `%` signs in a row. Comments to `MFT` can follow '`%%%%`'. Five or more `%` signs should not be used.

Beside the normal input file, `MFT` also looks for a change file (e.g., '`foo.ch`'), which allows substitutions to be made in the translation. The change file follows the conventions of `WEB`, and it should be null if there are no changes. (Changes usually contain verbatim instructions to compensate for the fact that `MFT` cannot format everything in an optimum way.)

There's also a third input file (e.g., '`plain.mft`'), which is input before the other two. This file normally contains the '`%%%`' formatting commands that are necessary to tune `MFT` to a particular style of METAFONT code, so it is called the style file.

The output of `MFT` should be accompanied by the macros in a small package called `mftmac.tex`.

Caveat: This program is not as "bulletproof" as the other routines produced by Stanford's TEX project. It takes care of a great deal of tedious formatting, but it can produce strange output, because METAFONT is an extremely general language. Users should proofread their output carefully.

2* `MFT` uses a few features of the local Pascal compiler that may need to be changed in other installations:

1) Case statements have a default.
2) Input-output routines may need to be adapted for use with a particular character set and/or for printing messages on the user's terminal.

These features are also present in the Pascal version of TEX, where they are used in a similar (but more complex) way. System-dependent portions of `MFT` can be identified by looking at the entries for 'system dependencies' in the index below.

The "banner line" defined here should be changed whenever `MFT` is modified.

define *my_name* ≡ ´`mft`´
define *banner* ≡ ´`This␣is␣MFT,␣Version␣2.1`´

3*. The program begins with a fairly normal header, made up of pieces that will mostly be filled in later. The MF input comes from files *mf_file*, *change_file*, and *style_file*; the TEX output goes to file *tex_file*.

If it is necessary to abort the job because of a fatal error, the program calls the '*jump_out*' procedure.

define *class* ≡ *class_var*

```
⟨ Compiler directives 4 ⟩
program MFT(mf_file, change_file, style_file, tex_file);
  const ⟨ Constants in the outer block 8* ⟩
  type ⟨ Types in the outer block 12 ⟩
  var ⟨ Globals in the outer block 9 ⟩
    ⟨ Error handling procedures 29 ⟩
    ⟨ Define parse_arguments 115* ⟩
  procedure initialize;
    var ⟨ Local variables for initialization 14 ⟩
    begin kpse_set_program_name(argv[0], my_name); parse_arguments; ⟨ Set initial values 10 ⟩;
    end;
```

8*. The following parameters are set big enough to handle the Computer Modern fonts, so they should be sufficient for most applications of MFT.

```
⟨ Constants in the outer block 8* ⟩ ≡
  max_bytes = 60000;   { the number of bytes in tokens; must be less than 65536 }
  max_names = 6000;   { number of tokens }
  hash_size = 353;   { should be prime }
  buf_size = 3000;   { maximum length of input line }
  line_length = 79;   { lines of TEX output have at most this many characters, should be less than 256 }
```

This code is used in section 3*.

13:* The original Pascal compiler was designed in the late 60s, when six-bit character sets were common, so it did not make provision for lowercase letters. Nowadays, of course, we need to deal with both capital and small letters in a convenient way, especially in a program for font design; so the present specification of MFT has been written under the assumption that the Pascal compiler and run-time system permit the use of text files with more than 64 distinguishable characters. More precisely, we assume that the character set contains at least the letters and symbols associated with ASCII codes *´40* through *´176*. If additional characters are present, MFT can be configured to work with them too.

Since we are dealing with more characters than were present in the first Pascal compilers, we have to decide what to call the associated data type. Some Pascals use the original name *char* for the characters in text files, even though there now are more than 64 such characters, while other Pascals consider *char* to be a 64-element subrange of a larger data type that has some other name.

In order to accommodate this difference, we shall use the name *text_char* to stand for the data type of the characters that are converted to and from *ASCII_code* when they are input and output. We shall also assume that *text_char* consists of the elements *chr*(*first_text_char*) through *chr*(*last_text_char*), inclusive. The following definitions should be adjusted if necessary.

```
define text_char ≡ ASCII_code  { the data type of characters in text files }
define first_text_char = 0  { ordinal number of the smallest element of text_char }
define last_text_char = 255  { ordinal number of the largest element of text_char }
```

⟨ Types in the outer block 12 ⟩ +≡

```
text_file = packed file of text_char;
```

17:* The ASCII code is "standard" only to a certain extent, since many computer installations have found it advantageous to have ready access to more than 94 printing characters. If MFT is being used on a garden-variety Pascal for which only standard ASCII codes will appear in the input and output files, it doesn't really matter what codes are specified in *xchr*[0 .. *´37*], but the safest policy is to blank everything out by using the code shown below.

However, other settings of *xchr* will make MFT more friendly on computers that have an extended character set, so that users can type things like '≠' instead of '<>', and so that MFT can echo the page breaks found in its input. People with extended character sets can assign codes arbitrarily, giving an *xchr* equivalent to whatever characters the users of MFT are allowed to have in their input files. Appropriate changes to MFT's *char_class* table should then be made. (Unlike TEX, each installation of METAFONT 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 10 ⟩ +≡

```
for i ← 1 to ´37 do xchr[i] ← chr(i);
for i ← ´177 to ´377 do xchr[i] ← chr(i);
```

20:* Terminal output is done by writing on file *term_out*, which is assumed to consist of characters of type *text_char*:

```
define term_out ≡ stdout
define print(#) ≡ write(term_out,#)  { 'print' means write on the terminal }
define print_ln(#) ≡ write_ln(term_out,#)  { 'print' and then start new line }
define new_line ≡ write_ln(term_out)  { start new line on the terminal }
define print_nl(#) ≡  { print information starting on a new line }
    begin new_line; print(#);
    end
```

21:* Different systems have different ways of specifying that the output on a certain file will appear on the user's terminal.

```
⟨ Set initial values 10 ⟩ +≡
  { nothing need be done }
```

22:* The *update_terminal* procedure 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.

```
define update_terminal ≡ fflush(term_out)  { empty the terminal output buffer }
```

24:* The following code opens the input files.

```
procedure open_input;  { prepare to read inputs }
  begin if metapost then mf_file ← kpse_open_file(cmdline(optind), kpse_mp_format)
  else mf_file ← kpse_open_file(cmdline(optind), kpse_mf_format);
  if change_name then
    begin if metapost then change_file ← kpse_open_file(change_name, kpse_mp_format)
    else change_file ← kpse_open_file(change_name, kpse_mf_format);
    end;
  style_file ← kpse_open_file(style_name[0], kpse_mft_format); i_style_name ← 1;
  end;
```

26:* The following code opens *tex_file*. Since this file was listed in the program header, we assume that the Pascal runtime system has checked that a suitable external file name has been given.

```
⟨ Set initial values 10 ⟩ +≡
  rewrite(tex_file, tex_name);
```

28:* The *input_ln* procedure brings the next line of input from the specified file into the *buffer* array and returns the value *true*, unless the file has already been entirely read, in which case it returns *false*. The conventions of TEX are followed; i.e., *ASCII_code* numbers representing the next line of the file are input into *buffer*[0], *buffer*[1], ..., *buffer*[*limit* − 1]; trailing blanks are ignored; and the global variable *limit* is set to the length of the line. The value of *limit* must be strictly less than *buf_size*.

```
function input_ln(var f : text_file): boolean;  { inputs a line or returns false }
  var final_limit: 0 .. buf_size;  { limit without trailing blanks }
  begin limit ← 0; final_limit ← 0;
  if eof(f) then input_ln ← false
  else begin while ¬eoln(f) do
      begin buffer[limit] ← xord[getc(f)]; incr(limit);
      if buffer[limit − 1] ≠ "␣" then final_limit ← limit;
      if limit = buf_size then
        begin while ¬eoln(f) do vgetc(f);
        decr(limit);  { keep buffer[buf_size] empty }
        if final_limit > limit then final_limit ← limit;
        print_nl('!␣Input␣line␣too␣long'); loc ← 0; error;
        end;
      end;
    read_ln(f); limit ← final_limit; input_ln ← true;
    end;
  end;
```

31.* The *jump_out* procedure cleans up, prints appropriate messages, and exits back to the operating system.

```
define fatal_error(#) ≡
        begin new_line; print(#); error; mark_fatal; jump_out;
        end
```

```
⟨ Error handling procedures 29 ⟩ +≡
procedure jump_out;
  begin { here files should be closed if the operating system requires it }
  ⟨ Print the job history 113 ⟩;
  new_line;
  if (history ≠ spotless) ∧ (history ≠ harmless_message) then uexit(1)
  else uexit(0);
  end;
```

47* ⟨ Read from *style_file* and maybe turn off *styling* 47* ⟩ ≡

```
begin incr(line);
if ¬input_ln(style_file) then
  begin if i_style_name ≠ n_style_name then
    begin xfclose(style_file, style_name[i_style_name − 1]);
    style_file ← kpse_open_file(style_name[i_style_name], kpse_mft_format);
    i_style_name ← i_style_name + 1;
    end
  else begin styling ← false;
    end;
  line ← 0;
  end;
end
```

This code is used in section 45.

63* Initializing the primitive tokens. Each token read by `MFT` is recognized as belonging to one of the following "types":

define *indentation* = 0 { internal code for space at beginning of a line }
define *end_of_line* = 1 { internal code for hypothetical token at end of a line }
define *end_of_file* = 2 { internal code for hypothetical token at end of the input }
define *verbatim* = 3 { internal code for the token '`%`' }
define *set_format* = 4 { internal code for the token '`%%`' }
define *mft_comment* = 5 { internal code for the token '`%%%`' }
define *min_action_type* = 6 { smallest code for tokens that produce "real" output }
define *numeric_token* = 6 { internal code for tokens like '`3.14159`' }
define *string_token* = 7 { internal code for tokens like '`"pie"`' }
define *min_symbolic_token* = 8 { smallest internal code for a symbolic token }
define *op* = 8 { internal code for tokens like '`sqrt`' }
define *command* = 9 { internal code for tokens like '`addto`' }
define *endit* = 10 { internal code for tokens like '`fi`' }
define *binary* = 11 { internal code for tokens like '`and`' }
define *abinary* = 12 { internal code for tokens like '`+`' }
define *bbinary* = 13 { internal code for tokens like '`step`' }
define *ampersand* = 14 { internal code for the token '`&`' }
define *pyth_sub* = 15 { internal code for the token '`+-+`' }
define *as_is* = 16 { internal code for tokens like '`]`' }
define *bold* = 17 { internal code for tokens like '`nullpen`' }
define *type_name* = 18 { internal code for tokens like '`numeric`' }
define *path_join* = 19 { internal code for the token '`..`' }
define *colon* = 20 { internal code for the token '`:`' }
define *semicolon* = 21 { internal code for the token '`;`' }
define *backslash* = 22 { internal code for the token '`\`' }
define *double_back* = 23 { internal code for the token '`\\`' }
define *less_or_equal* = 24 { internal code for the token '`<=`' }
define *greater_or_equal* = 25 { internal code for the token '`>=`' }
define *not_equal* = 26 { internal code for the token '`<>`' }
define *sharp* = 27 { internal code for the token '`#`' }
define *comment* = 28 { internal code for the token '`%`' }
define *recomment* = 29 { internal code used to resume a comment after '`|...|`' }
define *min_suffix* = 30 { smallest code for symbolic tokens in suffixes }
define *internal* = 30 { internal code for tokens like '`pausing`' }
define *input_command* = 31 { internal code for tokens like '`input`' }
define *btex_code* = 32 { begin TEX material (`btex`) }
define *verbatim_code* = 33 { begin TEX material (`verbatimtex`) }
define *etex_marker* = 34 { end TEX material (`etex`) }
define *special_tag* = 35 { internal code for tags that take at most one subscript }
define *tag* = 36 { internal code for nonprimitive tokens }

⟨ Assign the default value to *ilk*[*p*] 63* ⟩ ≡
ilk[*p*] ← *tag*

This code is used in section 62.

65:* We begin with primitives common to METAFONT and METAPOST.

The intended use of the macros above might not be immediately obvious, but the riddle is answered by the following:

⟨ Store all the primitives 65* ⟩ ≡

```
id_loc ← 18;
pr2("."")(".")(path_join);
pr1("[")(as_is);
pr1("]")(as_is);
pr1("}")(as_is);
pr1("{")(as_is);
pr1(":")(colon);
pr2(":")(":")(colon);
pr3("|")("|")(":")(colon);
pr2(":")("=")(as_is);
pr1(",")(as_is);
pr1(";")(semicolon);
pr1("\")(backslash);
pr2("\")("\")(double_back);
pr5("a")("d")("d")("t")("o")(command);
pr2("a")("t")(bbinary);
pr7("a")("t")("l")("e")("a")("s")("t")(op);
pr10("b")("e")("g")("i")("n")("g")("r")("o")("u")("p")(command);
pr8("c")("o")("n")("t")("r")("o")("l")("s")(op);
pr4("c")("u")("l")("l")(command);
pr4("c")("u")("r")("l")(op);
pr10("d")("e")("l")("i")("m")("i")("t")("e")("r")("s")(command);
pr8("e")("n")("d")("g")("r")("o")("u")("p")(endit);
pr8("e")("v")("e")("r")("y")("j")("o")("b")(command);
pr6("e")("x")("i")("t")("i")("f")(command);
pr11("e")("x")("p")("a")("n")("d")("a")("f")("t")("e")("r")(command);
pr4("f")("r")("o")("m")(bbinary);
pr7("i")("n")("t")("e")("r")("i")("m")(command);
pr3("l")("e")("t")(command);
pr11("n")("e")("w")("i")("n")("t")("e")("r")("n")("a")("l")(command);
pr2("o")("f")(command);
pr10("r")("a")("n")("d")("o")("m")("s")("e")("e")("d")(command);
pr4("s")("a")("v")("e")(command);
pr10("s")("c")("a")("n")("t")("o")("k")("e")("n")("s")(command);
pr7("s")("h")("i")("p")("o")("u")("t")(command);
pr4("s")("t")("e")("p")(bbinary);
pr3("s")("t")("r")(command);
pr7("t")("e")("n")("s")("i")("o")("n")(op);
pr2("t")("o")(bbinary);
pr5("u")("n")("t")("i")("l")(bbinary);
pr3("d")("e")("f")(command);
pr6("v")("a")("r")("d")("e")("f")(command);
```

See also sections 66*, 67*, 68*, 69*, 70*, 71*, 123*, and 124*.

This code is used in section 112*.

66:* (There are so many primitives, it's necessary to break this long initialization code up into pieces so as not to overflow `WEAVE`'s capacity.)

⟨ Store all the primitives 65* ⟩ +≡

pr10(`"p"`)(`"r"`)(`"i"`)(`"m"`)(`"a"`)(`"r"`)(`"y"`)(`"d"`)(`"e"`)(`"f"`)(*command*);
pr12(`"s"`)(`"e"`)(`"c"`)(`"o"`)(`"n"`)(`"d"`)(`"a"`)(`"r"`)(`"y"`)(`"d"`)(`"e"`)(`"f"`)(*command*);
pr11(`"t"`)(`"e"`)(`"r"`)(`"t"`)(`"i"`)(`"a"`)(`"r"`)(`"y"`)(`"d"`)(`"e"`)(`"f"`)(*command*);
pr6(`"e"`)(`"n"`)(`"d"`)(`"d"`)(`"e"`)(`"f"`)(*endit*);
pr3(`"f"`)(`"o"`)(`"r"`)(*command*);
pr11(`"f"`)(`"o"`)(`"r"`)(`"s"`)(`"u"`)(`"f"`)(`"f"`)(`"i"`)(`"x"`)(`"e"`)(`"s"`)(*command*);
pr7(`"f"`)(`"o"`)(`"r"`)(`"e"`)(`"v"`)(`"e"`)(`"r"`)(*command*);
pr6(`"e"`)(`"n"`)(`"d"`)(`"f"`)(`"o"`)(`"r"`)(*endit*);
pr5(`"q"`)(`"u"`)(`"o"`)(`"t"`)(`"e"`)(*command*);
pr4(`"e"`)(`"x"`)(`"p"`)(`"r"`)(*command*);
pr6(`"s"`)(`"u"`)(`"f"`)(`"f"`)(`"i"`)(`"x"`)(*command*);
pr4(`"t"`)(`"e"`)(`"x"`)(`"t"`)(*command*);
pr7(`"p"`)(`"r"`)(`"i"`)(`"m"`)(`"a"`)(`"r"`)(`"y"`)(*command*);
pr9(`"s"`)(`"e"`)(`"c"`)(`"o"`)(`"n"`)(`"d"`)(`"a"`)(`"r"`)(`"y"`)(*command*);
pr8(`"t"`)(`"e"`)(`"r"`)(`"t"`)(`"i"`)(`"a"`)(`"r"`)(`"y"`)(*command*);
pr5(`"i"`)(`"n"`)(`"p"`)(`"u"`)(`"t"`)(*input_command*);
pr8(`"e"`)(`"n"`)(`"d"`)(`"i"`)(`"n"`)(`"p"`)(`"u"`)(`"t"`)(*bold*);
pr2(`"i"`)(`"f"`)(*command*);
pr2(`"f"`)(`"i"`)(*endit*);
pr4(`"e"`)(`"l"`)(`"s"`)(`"e"`)(*command*);
pr6(`"e"`)(`"l"`)(`"s"`)(`"e"`)(`"i"`)(`"f"`)(*command*);
pr4(`"t"`)(`"r"`)(`"u"`)(`"e"`)(*bold*);
pr5(`"f"`)(`"a"`)(`"l"`)(`"s"`)(`"e"`)(*bold*);
pr11(`"n"`)(`"u"`)(`"l"`)(`"l"`)(`"p"`)(`"i"`)(`"c"`)(`"t"`)(`"u"`)(`"r"`)(`"e"`)(*bold*);
pr7(`"n"`)(`"u"`)(`"l"`)(`"l"`)(`"p"`)(`"e"`)(`"n"`)(*bold*);
pr7(`"j"`)(`"o"`)(`"b"`)(`"n"`)(`"a"`)(`"m"`)(`"e"`)(*bold*);
pr10(`"r"`)(`"e"`)(`"a"`)(`"d"`)(`"s"`)(`"t"`)(`"r"`)(`"i"`)(`"n"`)(`"g"`)(*bold*);
pr9(`"p"`)(`"e"`)(`"n"`)(`"c"`)(`"i"`)(`"r"`)(`"c"`)(`"l"`)(`"e"`)(*bold*);
pr2(`"="`)(`":"`)(*as_is*);
pr3(`"="`)(`":"`)(`"|"`)(*as_is*);
pr4(`"="`)(`":"`)(`"|"`)(`">"`)(*as_is*);
pr3(`"|"`)(`"="`)(`":"`)(*as_is*);
pr4(`"|"`)(`"="`)(`":"`)(`">"`)(*as_is*);
pr4(`"|"`)(`"="`)(`":"`)(`"|"`)(*as_is*);
pr5(`"|"`)(`"="`)(`":"`)(`"|"`)(`">"`)(*as_is*);
pr6(`"|"`)(`"="`)(`":"`)(`"|"`)(`">"`)(`">"`)(*as_is*);
pr4(`"k"`)(`"e"`)(`"r"`)(`"n"`)(*binary*); *pr6*(`"s"`)(`"k"`)(`"i"`)(`"p"`)(`"t"`)(`"o"`)(*command*);

67:* (Does anybody out there remember the commercials that went `LS-MFT`?)

⟨ Store all the primitives 65* ⟩ +≡

```
pr13("n")("o")("r")("m")("a")("l")("d")("e")("v")("i")("a")("t")("e")(op);
pr3("o")("d")("d")(op);
pr5("k")("n")("o")("w")("n")(op);
pr7("u")("n")("k")("n")("o")("w")("n")(op);
pr3("n")("o")("t")(op);
pr7("d")("e")("c")("i")("m")("a")("l")(op);
pr7("r")("e")("v")("e")("r")("s")("e")(op);
pr8("m")("a")("k")("e")("p")("a")("t")("h")(op);
pr7("m")("a")("k")("e")("p")("e")("n")(op);
pr3("o")("c")("t")(op);
pr3("h")("e")("x")(op);
pr5("A")("S")("C")("I")("I")(op);
pr4("c")("h")("a")("r")(op);
pr6("l")("e")("n")("g")("t")("h")(op);
pr13("t")("u")("r")("n")("i")("n")("g")("n")("u")("m")("b")("e")("r")(op);
pr5("x")("p")("a")("r")("t")(op);
pr5("y")("p")("a")("r")("t")(op);
pr6("x")("x")("p")("a")("r")("t")(op);
pr6("x")("y")("p")("a")("r")("t")(op);
pr6("y")("x")("p")("a")("r")("t")(op);
pr6("y")("y")("p")("a")("r")("t")(op);
pr4("s")("q")("r")("t")(op);
pr4("m")("e")("x")("p")(op);
pr4("m")("l")("o")("g")(op);
pr4("s")("i")("n")("d")(op);
pr4("c")("o")("s")("d")(op);
pr5("f")("l")("o")("o")("r")(op);
pr14("u")("n")("i")("f")("o")("r")("m")("d")("e")("v")("i")("a")("t")("e")(op);
pr10("c")("h")("a")("r")("e")("x")("i")("s")("t")("s")(op);
pr5("a")("n")("g")("l")("e")(op);
pr5("c")("y")("c")("l")("e")(op);
```

68:* (If you think this `WEB` code is ugly, you should see the Pascal code it produces.)

⟨ Store all the primitives 65* ⟩ +≡

```
pr13("t")("r")("a")("c")("i")("n")("g")("t")("i")("t")("l")("e")("s")(internal);
pr16("t")("r")("a")("c")("i")("n")("g")("e")("q")("u")("a")("t")("i")("o")("n")("s")(internal);
pr15("t")("r")("a")("c")("i")("n")("g")("c")("a")("p")("s")("u")("l")("e")("s")(internal);
pr14("t")("r")("a")("c")("i")("n")("g")("c")("h")("o")("i")("c")("e")("s")(internal);
pr12("t")("r")("a")("c")("i")("n")("g")("s")("p")("e")("c")("s")(internal);
pr11("t")("r")("a")("c")("i")("n")("g")("p")("e")("n")("s")(internal);
pr15("t")("r")("a")("c")("i")("n")("g")("c")("o")("m")("m")("a")("n")("d")("s")(internal);
pr13("t")("r")("a")("c")("i")("n")("g")("m")("a")("c")("r")("o")("s")(internal);
pr13("t")("r")("a")("c")("i")("n")("g")("o")("u")("t")("p")("u")("t")(internal);
pr12("t")("r")("a")("c")("i")("n")("g")("s")("t")("a")("t")("s")(internal);
pr13("t")("r")("a")("c")("i")("n")("g")("o")("n")("l")("i")("n")("e")(internal);
pr15("t")("r")("a")("c")("i")("n")("g")("r")("e")("s")("t")("o")("r")("e")("s")(internal);
```

69.* ⟨ Store all the primitives 65* ⟩ +≡

```
pr4("y")("e")("a")("r")(internal);
pr5("m")("o")("n")("t")("h")(internal);
pr3("d")("a")("y")(internal);
pr4("t")("i")("m")("e")(internal);
pr8("c")("h")("a")("r")("c")("o")("d")("e")(internal);
pr7("c")("h")("a")("r")("e")("x")("t")(internal);
pr6("c")("h")("a")("r")("w")("d")(internal);
pr6("c")("h")("a")("r")("h")("t")(internal);
pr6("c")("h")("a")("r")("d")("p")(internal);
pr6("c")("h")("a")("r")("i")("c")(internal);
pr6("c")("h")("a")("r")("d")("x")(internal);
pr6("c")("h")("a")("r")("d")("y")(internal);
pr10("d")("e")("s")("i")("g")("n")("s")("i")("z")("e")(internal);
pr7("x")("o")("f")("f")("s")("e")("t")(internal);
pr7("y")("o")("f")("f")("s")("e")("t")(internal);
pr7("p")("a")("u")("s")("i")("n")("g")(internal);
pr12("s")("h")("o")("w")("s")("t")("o")("p")("p")("i")("n")("g")(internal);
pr10("f")("o")("n")("t")("m")("a")("k")("i")("n")("g")(internal);
pr8("p")("r")("o")("o")("f")("i")("n")("g")(internal);
pr12("t")("u")("r")("n")("i")("n")("g")("c")("h")("e")("c")("k")(internal);
pr12("w")("a")("r")("n")("i")("n")("g")("c")("h")("e")("c")("k")(internal);
pr12("b")("o")("u")("n")("d")("a")("r")("y")("c")("h")("a")("r")(internal);
```

70:* Still more.

⟨ Store all the primitives 65* ⟩ +≡

```
pr1("+")(abinary);
pr1("-")(abinary);
pr1("*")(abinary);
pr1("/")(as_is);
pr2("+")("+")(binary);
pr3("+")("-")("+")(pyth_sub);
pr3("a")("n")("d")(binary);
pr2("o")("r")(binary);
pr1("<")(as_is);
pr2("<")("=")(less_or_equal);
pr1(">")(as_is);
pr2(">")("=")(greater_or_equal);
pr1("=")(as_is);
pr2("<")(">")(not_equal);
pr9("s")("u")("b")("s")("t")("r")("i")("n")("g")(command);
pr7("s")("u")("b")("p")("a")("t")("h")(command);
pr13("d")("i")("r")("e")("c")("t")("i")("o")("n")("t")("i")("m")("e")(command);
pr5("p")("o")("i")("n")("t")(command);
pr10("p")("r")("e")("c")("o")("n")("t")("r")("o")("l")(command);
pr11("p")("o")("s")("t")("c")("o")("n")("t")("r")("o")("l")(command);
pr9("p")("e")("n")("o")("f")("f")("s")("e")("t")(command);
pr1("&")(ampersand);
pr7("r")("o")("t")("a")("t")("e")("d")(binary);
pr7("s")("l")("a")("n")("t")("e")("d")(binary);
pr6("s")("c")("a")("l")("e")("d")(binary);
pr7("s")("h")("i")("f")("t")("e")("d")(binary);
pr11("t")("r")("a")("n")("s")("f")("o")("r")("m")("e")("d")(binary);
pr7("x")("s")("c")("a")("l")("e")("d")(binary);
pr7("y")("s")("c")("a")("l")("e")("d")(binary);
pr7("z")("s")("c")("a")("l")("e")("d")(binary);
pr17("i")("n")("t")("e")("r")("s")("e")("c")("t")("i")("o")("n")("t")("i")("m")("e")("s")(binary);
pr7("n")("u")("m")("e")("r")("i")("c")(type_name);
pr6("s")("t")("r")("i")("n")("g")(type_name);
pr7("b")("o")("o")("l")("e")("a")("n")(type_name);
pr4("p")("a")("t")("h")(type_name);
pr3("p")("e")("n")(type_name);
pr7("p")("i")("c")("t")("u")("r")("e")(type_name);
pr9("t")("r")("a")("n")("s")("f")("o")("r")("m")(type_name);
pr4("p")("a")("i")("r")(type_name);
```

71:* At last we are done with the tedious initialization of primitives.

⟨ Store all the primitives 65* ⟩ +≡

```
pr3("e")("n")("d")(endit);
pr4("d")("u")("m")("p")(endit);
pr9("b")("a")("t")("c")("h")("m")("o")("d")("e")(bold);
pr11("n")("o")("n")("s")("t")("o")("p")("m")("o")("d")("e")(bold);
pr10("s")("c")("r")("o")("l")("l")("m")("o")("d")("e")(bold);
pr13("e")("r")("r")("o")("r")("s")("t")("o")("p")("m")("o")("d")("e")(bold);
pr5("i")("n")("n")("e")("r")(command);
pr5("o")("u")("t")("e")("r")(command);
pr9("s")("h")("o")("w")("t")("o")("k")("e")("n")(command);
pr9("s")("h")("o")("w")("s")("t")("a")("t")("s")(bold);
pr4("s")("h")("o")("w")(command);
pr12("s")("h")("o")("w")("v")("a")("r")("i")("a")("b")("l")("e")(command);
pr16("s")("h")("o")("w")("d")("e")("p")("e")("n")("d")("e")("n")("c")("i")("e")("s")(bold);
pr7("c")("o")("n")("t")("o")("u")("r")(command);
pr10("d")("o")("u")("b")("l")("e")("p")("a")("t")("h")(command);
pr4("a")("l")("s")("o")(command);
pr7("w")("i")("t")("h")("p")("e")("n")(command);
pr7("m")("e")("s")("s")("a")("g")("e")(command);
pr10("e")("r")("r")("m")("e")("s")("s")("a")("g")("e")(command);
pr7("e")("r")("r")("h")("e")("l")("p")(command);
pr8("c")("h")("a")("r")("l")("i")("s")("t")(command);
pr8("l")("i")("g")("t")("a")("b")("l")("e")(command);
pr10("e")("x")("t")("e")("n")("s")("i")("b")("l")("e")(command);
pr10("h")("e")("a")("d")("e")("r")("b")("y")("t")("e")(command);
pr9("f")("o")("n")("t")("d")("i")("m")("e")("n")(command);
pr7("s")("p")("e")("c")("i")("a")("l")(command);
pr1("%")(comment);
pr2("%")("%")(verbatim);
pr3("%")("%")("%")(set_format);
pr4("%")("%")("%")("%")(mft_comment);
pr1("#")(sharp);
pr4("g")("o")("o")("d")(special_tag);
```

75:* Inputting the next token. MFT's lexical scanning routine is called *get_next*. This procedure inputs the next token of METAFONT input and puts its encoded meaning into two global variables, *cur_type* and *cur_tok*.

The **btex** ... **etex** and **verbatimtex** ... **etex** features need to be implemented at a low level in the scanning process. This is implemented by changing the behavior of the scanner via *scanner_status* global variable.

define *normal* = 0 { *scanner_status* at "quiet times" }
define *verbatimtex_flushing* = 1 { *scanner_status* when moving verbatim TEX material }
define *btex_flushing* = 2 { *scanner_status* when moving TEX code }

⟨ Globals in the outer block 9 ⟩ +≡
cur_type: *eight_bits*; { type of token just scanned }
cur_tok: *integer*; { hash table or buffer location }
prev_type: *eight_bits*; { previous value of *cur_type* }
prev_tok: *integer*; { previous value of *cur_tok* }
scanner_status: *normal* .. *btex_flushing*; { are we scanning at high speed? }

79:* If changes are made to accommodate non-ASCII character sets, they should be essentially the same in MFT as in METAFONT. However, MFT has an additional class number, the *end_line_class*, which is used only for the special character *carriage_return* that is placed at the end of the input buffer.

define *carriage_return* = ´15 { special code placed in *buffer*[*limit*] }

⟨ Set initial values 10 ⟩ +≡
for *i* ← "0" **to** "9" **do** *char_class*[*i*] ← *digit_class*;
char_class["."] ← *period_class*; *char_class*["␣"] ← *space_class*; *char_class*["%"] ← *percent_class*;
char_class[""""] ← *string_class*;
char_class[","] ← 5; *char_class*[";"] ← 6; *char_class*["("] ← 7; *char_class*[")"] ← *right_paren_class*;
for *i* ← "A" **to** "Z" **do** *char_class*[*i*] ← *letter_class*;
for *i* ← "a" **to** "z" **do** *char_class*[*i*] ← *letter_class*;
char_class["_"] ← *letter_class*;
char_class["<"] ← 10; *char_class*["="] ← 10; *char_class*[">"] ← 10; *char_class*[":"] ← 10;
char_class["|"] ← 10;
char_class["`"] ← 11; *char_class*["´"] ← 11;
char_class["+"] ← 12; *char_class*["-"] ← 12;
char_class["/"] ← 13; *char_class*["*"] ← 13; *char_class*["\"] ← 13;
char_class["!"] ← 14; *char_class*["?"] ← 14;
char_class["#"] ← 15; *char_class*["&"] ← 15; *char_class*["@"] ← 15; *char_class*["$"] ← 15;
char_class["^"] ← 16; *char_class*["~"] ← 16;
char_class["["] ← *left_bracket_class*; *char_class*["]"] ← *right_bracket_class*;
char_class["{"] ← 19; *char_class*["}"] ← 19;
for *i* ← 0 **to** "␣" − 1 **do** *char_class*[*i*] ← *letter_class*;
for *i* ← 127 **to** 255 **do** *char_class*[*i*] ← *letter_class*;
char_class[*carriage_return*] ← *end_line_class*; *char_class*[´11] ← *space_class*; { tab }
char_class[´14] ← *space_class*; { form feed }

81.* **define** *emit*(#) ≡ **begin** *cur_type* ← #; *cur_tok* ← *id_first*; **return**; **end**

⟨ Branch on the *class*, scan the token; **return** directly if the token is special, or **goto** *found* if it needs to be looked up 81* ⟩ ≡

```
case class of
digit_class: goto pass_digits;
period_class: begin class ← char_class[buffer[loc]];
  if class > period_class then goto switch   { ignore isolated '.' }
  else if class < period_class then goto pass_fraction;   { class = digit_class }
  end;
space_class: if start_of_line ∨ scanner_status > normal then emit(indentation)
  else goto switch;
end_line_class: emit(end_of_line);
string_class: ⟨ Get a string token and return 82 ⟩;
isolated_classes: goto found;
invalid_class: ⟨ Decry the invalid character and goto switch 84 ⟩;
othercases do_nothing   { letters, etc. }
endcases;
while char_class[buffer[loc]] = class do incr(loc);
goto found;
pass_digits: while char_class[buffer[loc]] = digit_class do incr(loc);
  if buffer[loc] ≠ "." then goto done;
  if char_class[buffer[loc + 1]] ≠ digit_class then goto done;
  incr(loc);
pass_fraction: repeat incr(loc);
  until char_class[buffer[loc]] ≠ digit_class;
done: emit(numeric_token)
```

This code is used in section 80.

88:* MFT calls *flush_buffer*(*out_ptr*, *false*) before it has input anything. We initialize the output variables so that the first line of the output file will be '`\input mftmac`' or '`\input mptmac`' if a METAPOST file is converted.

```
⟨ Set initial values 10 ⟩ +≡
  out_ptr ← 1; out_buf[1] ← "␣"; out_line ← 1;
  if metapost then
    begin write(tex_file, ´\input␣mptmac´);
    end
  else write(tex_file, ´\input␣mftmac´);
```

97* Translation. The main work of MFT is accomplished by a routine that translates the tokens, one by one, with a limited amount of lookahead/lookbehind. Automata theorists might loosely call this a "finite state transducer," because the flow of control is comparatively simple.

```
procedure do_the_translation;
  label restart, reswitch, done, exit;
  var k: 0 .. buf_size;  { looks ahead in the buffer }
    t: integer;  { type that spreads to new tokens }
  begin restart: if out_ptr > 0 then flush_buffer(out_ptr, false);
  empty_buffer ← true;
  loop begin get_next;
    if start_of_line then ⟨ Do special actions at the start of a line 98 ⟩;
  reswitch: case cur_type of
    numeric_token: ⟨ Translate a numeric token or a fraction 105 ⟩;
    string_token: ⟨ Translate a string token 99 ⟩;
    verbatim_code, btex_code: ⟨ Copy TEX material 125* ⟩;
    indentation: out_str(tr_quad);
    end_of_line, mft_comment: ⟨ Wind up a line of translation and goto restart, or finish a | ... | segment
        and goto reswitch 110 ⟩;
    end_of_file: return;
    ⟨ Cases that translate primitive tokens 100 ⟩
    comment, recomment: ⟨ Translate a comment and goto restart, unless there's a | ... | segment 108 ⟩;
    verbatim: ⟨ Copy the rest of the current input line to the output, then goto restart 109 ⟩;
    set_format: ⟨ Change the translation format of tokens, and goto restart or reswitch 111 ⟩;
    internal, special_tag, tag: ⟨ Translate a tag and possible subscript 106 ⟩;
    end;  { all cases have been listed }
    end;
exit: end;
```

112:* The main program. Let's put it all together now: `MFT` starts and ends here.

```
begin initialize; { beginning of the main program }
print(banner); { print a "banner line" }
print_ln(version_string); ⟨ Store all the primitives 65* ⟩;
⟨ Store all the translations 73 ⟩;
⟨ Initialize the input system 44 ⟩;
do_the_translation; ⟨ Check that all changes have been read 49 ⟩;
⟨ Print the job history 113 ⟩;
new_line;
if (history ≠ spotless) ∧ (history ≠ harmless_message) then uexit(1)
else uexit(0);
end.
```

114.* System-dependent changes. The user calls `MFT` with arguments on the command line. These are either filenames or flags (beginning with '`-`'). The following globals are for communicating the user's desires to the rest of the program. The various *name* variables contain strings with the full names of those files, as UNIX knows them.

define *max_style_name* = 32

```
⟨ Globals in the outer block 9 ⟩ +≡
change_name, tex_name: const_c_string;
style_name: array [0 .. max_style_name − 1] of const_c_string;
n_style_name: c_int_type; { Number of values in style_name array. }
i_style_name: c_int_type; { The next style_name. }
metapost: c_int_type; { true for METAPOST, false for METAFONT }
```

115:* Look at the command line arguments and set the *name* variables accordingly.

At least one file name must be present as the first argument: the `mf` file. It may have an extension, or it may omit it to get `´.mf´` added. If there is only one file name, the output file name is formed by replacing the `mf` file name extension by `´.tex´`. Thus, the command line `mf foo` implies the use of the METAFONT input file `foo.mf` and the output file `foo.tex`. If this style of command line, with only one argument, is used, the default style file, *plain.mft*, will be used to provide basic formatting.

define *argument_is*(#) ≡ (*strcmp*(*long_options*[*option_index*].*name*, #) = 0)

```
⟨ Define parse_arguments 115* ⟩ ≡
procedure parse_arguments;
  const n_options = 5;  { Pascal won't count array lengths for us. }
  var long_options: array [0 .. n_options] of getopt_struct;
    getopt_return_val: integer; option_index: c_int_type; current_option: 0 .. n_options;
    suffix: const_c_string;
  begin ⟨ Initialize the option variables 121* ⟩;
  ⟨ Define the option table 116* ⟩;
  n_style_name ← 0;
  repeat getopt_return_val ← getopt_long_only(argc, argv, ´´, long_options, address_of(option_index));
    if getopt_return_val = −1 then
      begin do_nothing;  { End of arguments; we exit the loop below. }
      end
    else if getopt_return_val = "?" then
        begin usage(my_name);
        end
      else if argument_is(´help´) then
          begin usage_help(MFT_HELP, nil);
          end
        else if argument_is(´version´) then
            begin print_version_and_exit(banner, nil, ´D.E.␣Knuth␣␣(MP␣changes␣by␣W.␣Bzyl)´, nil);
            end
          else if argument_is(´change´) then
              begin change_name ← extend_filename(optarg, ´ch´);
              end
            else if argument_is(´style´) then
                begin if (n_style_name = max_style_name) then
                  begin fatal_error(´Too␣many␣style␣files␣specified.´); usage(my_name);
                  end;
                style_name[n_style_name] ← extend_filename(optarg, ´mft´);
                n_style_name ← n_style_name + 1;
                end;  { Else it was a flag; getopt has already done the assignment. }
  until getopt_return_val = −1;  { Now optind is the index of first non-option on the command line. We
        must have exactly one remaining argument. }
  if (optind + 1 ≠ argc) then
    begin write_ln(stderr, my_name, ´:␣Need␣exactly␣one␣file␣argument.´); usage(my_name);
    end;
  suffix ← find_suffix(cmdline(optind));
  if suffix ∧ (strcmp(suffix, ´mp´) = 0) then
    begin metapost ← true; tex_name ← basename_change_suffix(cmdline(optind), ´.mp´, ´.tex´);
    end
  else begin tex_name ← basename_change_suffix(cmdline(optind), ´.mf´, ´.tex´);
    end;
  if (n_style_name = 0) then
    begin if metapost then style_name[0] ← ´mplain.mft´
```

```
    else style_name[0] ← ´plain.mft´;
    n_style_name ← 1;
    end;
  end;
```

This code is used in section 3*.

116.* Here are the options we allow. The first is one of the standard GNU options.

```
⟨ Define the option table 116* ⟩ ≡
  current_option ← 0; long_options[current_option].name ← ´help´;
  long_options[current_option].has_arg ← 0; long_options[current_option].flag ← 0;
  long_options[current_option].val ← 0; incr(current_option);
```

See also sections 117*, 118*, 119*, 120*, and 122*.

This code is used in section 115*.

117.* Another of the standard options.

```
⟨ Define the option table 116* ⟩ +≡
  long_options[current_option].name ← ´version´; long_options[current_option].has_arg ← 0;
  long_options[current_option].flag ← 0; long_options[current_option].val ← 0; incr(current_option);
```

118.* Here is the option to set a change file.

```
⟨ Define the option table 116* ⟩ +≡
  long_options[current_option].name ← ´change´; long_options[current_option].has_arg ← 1;
  long_options[current_option].flag ← 0; long_options[current_option].val ← 0; incr(current_option);
```

119.* Here is the option to set the style file.

```
⟨ Define the option table 116* ⟩ +≡
  long_options[current_option].name ← ´style´; long_options[current_option].has_arg ← 1;
  long_options[current_option].flag ← 0; long_options[current_option].val ← 0; incr(current_option);
```

120.* The option to set a METAPOST file processing.

```
⟨ Define the option table 116* ⟩ +≡
  long_options[current_option].name ← ´metapost´; long_options[current_option].has_arg ← 0;
  long_options[current_option].flag ← address_of(metapost); long_options[current_option].val ← 1;
  incr(current_option);
```

121.* *metapost* defaults to *false*; will become *true* for METAPOST.

```
⟨ Initialize the option variables 121* ⟩ ≡
  metapost ← false;
```

This code is used in section 115*.

122.* An element with all zeros always ends the list of options.

```
⟨ Define the option table 116* ⟩ +≡
  long_options[current_option].name ← 0; long_options[current_option].has_arg ← 0;
  long_options[current_option].flag ← 0; long_options[current_option].val ← 0;
```

123:* Store primitives specific for METAFONT.

⟨ Store all the primitives 65* ⟩ +≡

```
if ¬metapost then
  begin pr12("a")("u")("t")("o")("r")("o")("u")("n")("d")("i")("n")("g")(internal);
  pr7("c")("h")("a")("r")("f")("a")("m")(internal);
  pr8("d")("r")("o")("p")("p")("i")("n")("g")(command);
  pr7("d")("i")("s")("p")("l")("a")("y")(command);
  pr6("f")("i")("l")("l")("i")("n")(internal);
  pr11("g")("r")("a")("n")("u")("l")("a")("r")("i")("t")("y")(internal);
  pr8("i")("n")("w")("i")("n")("d")("o")("w")(bbinary);
  pr4("h")("p")("p")("p")(internal);
  pr7("k")("e")("e")("p")("i")("n")("g")(command);
  pr10("n")("u")("m")("s")("p")("e")("c")("i")("a")("l")(command);
  pr10("o")("p")("e")("n")("w")("i")("n")("d")("o")("w")(command);
  pr9("s")("m")("o")("o")("t")("h")("i")("n")("g")(internal);
  pr4("v")("p")("p")("p")(internal);
  pr11("t")("o")("t")("a")("l")("w")("e")("i")("g")("h")("t")(op);
  pr12("t")("r")("a")("c")("i")("n")("g")("e")("d")("g")("e")("s")(internal);
  pr10("w")("i")("t")("h")("w")("e")("i")("g")("h")("t")(command);
  end;
```

124:* Store primitives specific for METAPOST.

```
⟨ Store all the primitives 65* ⟩ +≡
  if metapost then
    begin pr9("a")("r")("c")("l")("e")("n")("g")("t")("h")(op);
    pr7("a")("r")("c")("t")("i")("m")("e")(command);
    pr8("b")("l")("u")("e")("p")("a")("r")("t")(op);
    pr7("b")("o")("u")("n")("d")("e")("d")(op);
    pr4("b")("t")("e")("x")(btex_code);
    pr4("c")("l")("i")("p")(command);
    pr7("c")("l")("i")("p")("p")("e")("d")(op);
    pr9("c")("l")("o")("s")("e")("f")("r")("o")("m")(input_command);
    pr5("c")("o")("l")("o")("r")(type_name);
    pr6("d")("a")("s")("h")("e")("d")(command);
    pr8("d")("a")("s")("h")("p")("a")("r")("t")(op);
    pr4("e")("t")("e")("x")(etex_marker);
    pr6("f")("i")("l")("l")("e")("d")(op);
    pr8("f")("o")("n")("t")("p")("a")("r")("t")(op);
    pr8("f")("o")("n")("t")("s")("i")("z")("e")(op);
    pr9("g")("r")("e")("e")("n")("p")("a")("r")("t")(op);
    pr6("i")("n")("f")("o")("n")("t")(binary);
    pr8("l")("i")("n")("e")("j")("o")("i")("n")(internal);
    pr7("l")("i")("n")("e")("c")("a")("p")(internal);
    pr8("l")("l")("c")("o")("r")("n")("e")("r")(op);
    pr8("l")("r")("c")("o")("r")("n")("e")("r")(op);
    pr10("m")("i")("t")("e")("r")("l")("i")("m")("i")("t")(internal);
    pr8("m")("p")("x")("b")("r")("e")("a")("k")(bold);
    pr8("p")("a")("t")("h")("p")("a")("r")("t")(op);
    pr7("p")("e")("n")("p")("a")("r")("t")(op);
    pr9("p")("r")("o")("l")("o")("g")("u")("e")("s")(internal);
    pr7("r")("e")("d")("p")("a")("r")("t")(op);
    pr8("r")("e")("a")("d")("f")("r")("o")("m")(input_command);
    pr9("s")("e")("t")("b")("o")("u")("n")("d")("s")(command);
    pr7("s")("t")("r")("o")("k")("e")("d")(op);
    pr8("t")("e")("x")("t")("p")("a")("r")("t")(op);
    pr7("t")("e")("x")("t")("u")("a")("l")(op);
    pr16("t")("r")("a")("c")("i")("n")("g")("l")("o")("s")("t")("c")("h")("a")("r")("s")(internal);
    pr11("t")("r")("u")("e")("c")("o")("r")("n")("e")("r")("s")(internal);
    pr8("u")("l")("c")("o")("r")("n")("e")("r")(op);
    pr8("u")("r")("c")("o")("r")("n")("e")("r")(op);
    pr11("v")("e")("r")("b")("a")("t")("i")("m")("t")("e")("x")(verbatim_code);
    pr6("w")("i")("t")("h")("i")("n")(bbinary);
    pr9("w")("i")("t")("h")("c")("o")("l")("o")("r")(command);
    pr5("w")("r")("i")("t")("e")(command);
    end;
```

125:* Here an extra section is added. '`btex`' token is translated to '`\mftbeginB`' and '`verbatimtex`' to '`\mftbeginV`'. '`etex`' is translated to '`\mftend`'. These TEX macros are defined in `mptmac.tex`.

```
⟨ Copy TEX material 125* ⟩ ≡
  begin out4("\")("m")("f")("t"); out5("b")("e")("g")("i")("n");
  if cur_type = verbatim_code then
    begin out("V"); scanner_status ← verbatimtex_flushing;
    end
  else if cur_type = btex_code then
      begin out("B"); scanner_status ← btex_flushing;
      end;
  out("{"); out_name(cur_tok); out("}"); get_next;
  while cur_type ≠ etex_marker do
    begin if cur_type = indentation then
      begin out("␣");
      end
    else if cur_type = end_of_line then
        begin flush_buffer(out_ptr, false); empty_buffer ← true;
        end
      else copy(id_first);
    get_next;
    end;
  out4("\")("m")("f")("t"); out3("e")("n")("d"); out("{"); out_name(cur_tok); out("}");
  if scanner_status = verbatimtex_flushing then out("$");
  scanner_status ← normal;
  end
```

This code is used in section 97*.

126.* Index.

The following sections were changed by the change file: 1, 2, 3, 8, 13, 17, 20, 21, 22, 24, 26, 28, 31, 47, 63, 65, 66, 67, 68, 69, 70, 71, 75, 79, 81, 88, 97, 112, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126.