The MFT processor

(Version 2.0, October 1989)

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 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.

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 *banner* ≡ ´This␣is␣MFT,␣Version␣2.0´

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, which goes to the label *end_of_MFT*.

define *end_of_MFT* = 9999 { go here to wrap it up }

```
⟨ Compiler directives 4 ⟩
program MFT(mf_file, change_file, style_file, tex_file);
  label end_of_MFT; { go here to finish }
  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 ⟩
  procedure initialize;
    var ⟨ Local variables for initialization 14 ⟩
    begin ⟨ Set initial values 10 ⟩
    end;
```

4. The Pascal compiler used to develop this system has "compiler directives" that can appear in comments whose first character is a dollar sign. In our case these directives tell the compiler to detect things that are out of range.

```
⟨ Compiler directives 4 ⟩ ≡
  @{@&$C+, A+, D−@}  { range check, catch arithmetic overflow, no debug overhead }
```

This code is used in section 3.

5. Labels are given symbolic names by the following definitions. We insert the label '*exit*:' just before the '**end**' of a procedure in which we have used the '**return**' statement defined below; the label '*restart*' is occasionally used at the very beginning of a procedure; and the label '*reswitch*' is occasionally used just prior to a **case** statement in which some cases change the conditions and we wish to branch to the newly applicable case. Loops that are set up with the **loop** construction defined below are commonly exited by going to '*done*' or to '*found*' or to '*not_found*', and they are sometimes repeated by going to '*continue*'.

```
define exit = 10  { go here to leave a procedure }
define restart = 20  { go here to start a procedure again }
define reswitch = 21  { go here to start a case statement again }
define continue = 22  { go here to resume a loop }
define done = 30  { go here to exit a loop }
define found = 31  { go here when you've found it }
define not_found = 32  { go here when you've found something else }
```

6. Here are some macros for common programming idioms.

```
define incr(#) ≡ # ← # + 1  { increase a variable by unity }
define decr(#) ≡ # ← # − 1  { decrease a variable by unity }
define loop ≡ while true do  { repeat over and over until a goto happens }
define do_nothing ≡  { empty statement }
define return ≡ goto exit  { terminate a procedure call }
format return ≡ nil
format loop ≡ xclause
```

7. We assume that **case** statements may include a default case that applies if no matching label is found. Thus, we shall use constructions like

```
case x of
1: ⟨ code for x = 1 ⟩;
3: ⟨ code for x = 3 ⟩;
othercases ⟨ code for x ≠ 1 and x ≠ 3 ⟩
endcases
```

since most Pascal compilers have plugged this hole in the language by incorporating some sort of default mechanism. For example, the compiler used to develop WEB and TEX allows '*others*:' as a default label, and other Pascals allow syntaxes like '**else**' or '**otherwise**' or '*otherwise*:', etc. The definitions of **othercases** and **endcases** should be changed to agree with local conventions. (Of course, if no default mechanism is available, the **case** statements of this program must be extended by listing all remaining cases.)

```
define othercases ≡ others:   { default for cases not listed explicitly }
define endcases ≡ end   { follows the default case in an extended case statement }
format othercases ≡ else
format endcases ≡ 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 = 10000;   { the number of bytes in tokens; must be less than 65536 }
  max_names = 1000;   { number of tokens }
  hash_size = 353;   { should be prime }
  buf_size = 100;   { maximum length of input line }
  line_length = 80;   { lines of TEX output have at most this many characters, should be less than 256 }
```

This code is used in section 3.

9. A global variable called *history* will contain one of four values at the end of every run: *spotless* means that no unusual messages were printed; *harmless_message* means that a message of possible interest was printed but no serious errors were detected; *error_message* means that at least one error was found; *fatal_message* means that the program terminated abnormally. The value of *history* does not influence the behavior of the program; it is simply computed for the convenience of systems that might want to use such information.

```
define spotless = 0   { history value for normal jobs }
define harmless_message = 1   { history value when non-serious info was printed }
define error_message = 2   { history value when an error was noted }
define fatal_message = 3   { history value when we had to stop prematurely }
define mark_harmless ≡ if history = spotless then history ← harmless_message
define mark_error ≡ history ← error_message
define mark_fatal ≡ history ← fatal_message
```

```
⟨ Globals in the outer block 9 ⟩ ≡
history: spotless .. fatal_message;   { how bad was this run? }
```

See also sections 15, 20, 23, 25, 27, 34, 36, 51, 53, 55, 72, 74, 75, 77, 78, and 86.

This code is used in section 3.

10. ⟨ Set initial values 10 ⟩ ≡

```
  history ← spotless;
```

See also sections 16, 17, 18, 21, 26, 54, 57, 76, 79, 88, and 90.

This code is used in section 3.

11. The character set. MFT works internally with ASCII codes, like all other programs associated with TEX and METAFONT. The present section has been lifted almost verbatim from the METAFONT program.

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

⟨ Types in the outer block 12 ⟩ ≡
  ASCII_code = 0 .. 255; { eight-bit numbers }

See also sections 13, 50, and 52.

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* ≡ *char* { 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*;

14. ⟨ Local variables for initialization 14 ⟩ ≡
i: 0 .. 255;

See also section 56.

This code is used in section 3.

15. The MFT 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.

⟨ Globals in the outer block 9 ⟩ +≡
xord: **array** [*text_char*] **of** *ASCII_code*; { specifies conversion of input characters }
xchr: **array** [*ASCII_code*] **of** *text_char*; { specifies conversion of output characters }

16. Since we are assuming that our Pascal system is able to read and write the visible characters of standard ASCII (although not necessarily using the ASCII codes to represent them), the following assignment statements initialize most of the *xchr* array properly, without needing any system-dependent changes. On the other hand, it is possible to implement MFT with less complete character sets, and in such cases it will be necessary to change something here.

⟨ Set initial values 10 ⟩ +≡

```
xchr['40] ← ' '; xchr['41] ← '!'; xchr['42] ← '"'; xchr['43] ← '#'; xchr['44] ← '$';
xchr['45] ← '%'; xchr['46] ← '&'; xchr['47] ← '''';
xchr['50] ← '('; xchr['51] ← ')'; xchr['52] ← '*'; xchr['53] ← '+'; xchr['54] ← ',';
xchr['55] ← '-'; xchr['56] ← '.'; xchr['57] ← '/';
xchr['60] ← '0'; xchr['61] ← '1'; xchr['62] ← '2'; xchr['63] ← '3'; xchr['64] ← '4';
xchr['65] ← '5'; xchr['66] ← '6'; xchr['67] ← '7';
xchr['70] ← '8'; xchr['71] ← '9'; xchr['72] ← ':'; xchr['73] ← ';'; xchr['74] ← '<';
xchr['75] ← '='; xchr['76] ← '>'; xchr['77] ← '?';
xchr['100] ← '@'; xchr['101] ← 'A'; xchr['102] ← 'B'; xchr['103] ← 'C'; xchr['104] ← 'D';
xchr['105] ← 'E'; xchr['106] ← 'F'; xchr['107] ← 'G';
xchr['110] ← 'H'; xchr['111] ← 'I'; xchr['112] ← 'J'; xchr['113] ← 'K'; xchr['114] ← 'L';
xchr['115] ← 'M'; xchr['116] ← 'N'; xchr['117] ← 'O';
xchr['120] ← 'P'; xchr['121] ← 'Q'; xchr['122] ← 'R'; xchr['123] ← 'S'; xchr['124] ← 'T';
xchr['125] ← 'U'; xchr['126] ← 'V'; xchr['127] ← 'W';
xchr['130] ← 'X'; xchr['131] ← 'Y'; xchr['132] ← 'Z'; xchr['133] ← '['; xchr['134] ← '\';
xchr['135] ← ']'; xchr['136] ← '^'; xchr['137] ← '_';
xchr['140] ← '`'; xchr['141] ← 'a'; xchr['142] ← 'b'; xchr['143] ← 'c'; xchr['144] ← 'd';
xchr['145] ← 'e'; xchr['146] ← 'f'; xchr['147] ← 'g';
xchr['150] ← 'h'; xchr['151] ← 'i'; xchr['152] ← 'j'; xchr['153] ← 'k'; xchr['154] ← 'l';
xchr['155] ← 'm'; xchr['156] ← 'n'; xchr['157] ← 'o';
xchr['160] ← 'p'; xchr['161] ← 'q'; xchr['162] ← 'r'; xchr['163] ← 's'; xchr['164] ← 't';
xchr['165] ← 'u'; xchr['166] ← 'v'; xchr['167] ← 'w';
xchr['170] ← 'x'; xchr['171] ← 'y'; xchr['172] ← 'z'; xchr['173] ← '{'; xchr['174] ← '|';
xchr['175] ← '}'; xchr['176] ← '~';
```

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 ← 0 to '37 do xchr[i] ← ' ';
for i ← '177 to '377 do xchr[i] ← ' ';
```

18. 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 $i < j < \acute{}177$, 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 *´40* in case there is a coincidence.

```
⟨ Set initial values 10 ⟩ +≡
  for i ← first_text_char to last_text_char do xord[chr(i)] ← ´177;
  for i ← ´200 to ´377 do xord[xchr[i]] ← i;
  for i ← 1 to ´176 do xord[xchr[i]] ← i;
```

19. Input and output. The I/O conventions of this program are essentially identical to those of WEAVE. Therefore people who need to make modifications should be able to do so without too many headaches.

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

```
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
```

⟨ Globals in the outer block 9 ⟩ +≡
term_out: *text_file*; { the terminal as an output file }

21. Different systems have different ways of specifying that the output on a certain file will appear on the user's terminal. Here is one way to do this on the Pascal system that was used in WEAVE's initial development:

⟨ Set initial values 10 ⟩ +≡
rewrite(*term_out*, 'TTY:'); { send *term_out* output to the terminal }

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 ≡ break(term_out)   { empty the terminal output buffer }
```

23. The main input comes from *mf_file*; this input may be overridden by changes in *change_file*. (If *change_file* is empty, there are no changes.) Furthermore the *style_file* is input first; it is unchangeable.

⟨ Globals in the outer block 9 ⟩ +≡
mf_file: *text_file*; { primary input }
change_file: *text_file*; { updates }
style_file: *text_file*; { formatting bootstrap }

24. The following code opens the input files. Since these files were listed in the program header, we assume that the Pascal runtime system has already checked that suitable file names have been given; therefore no additional error checking needs to be done.

```
procedure open_input;   { prepare to read the inputs }
  begin reset(mf_file); reset(change_file); reset(style_file);
  end;
```

25. The main output goes to *tex_file*.

⟨ Globals in the outer block 9 ⟩ +≡
tex_file: *text_file*;

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*);

27. Input goes into an array called *buffer*.

⟨ Globals in the outer block 9 ⟩ +≡
buffer: **array** [0 .. *buf_size*] **of** *ASCII_code*;

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[f↑]; get(f); incr(limit);
      if buffer[limit − 1] ≠ "␣" then final_limit ← limit;
      if limit = buf_size then
        begin while ¬eoln(f) do get(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;
```

29. Reporting errors to the user. The command '*err_print*(`! Error message`)' will report a syntax error to the user, by printing the error message at the beginning of a new line and then giving an indication of where the error was spotted in the source file. Note that no period follows the error message, since the error routine will automatically supply a period.

The actual error indications are provided by a procedure called *error*.

```
define err_print(#) ≡
        begin new_line; print(#); error;
        end
```

```
⟨ Error handling procedures 29 ⟩ ≡
procedure error; { prints '.' and location of error message }
  var k, l: 0 .. buf_size; { indices into buffer }
  begin ⟨ Print error location based on input buffer 30 ⟩;
  update_terminal; mark_error;
  end;
```

See also section 31.

This code is used in section 3.

30. The error locations can be indicated by using the global variables *loc*, *line*, *styling*, and *changing*, which tell respectively the first unlooked-at position in *buffer*, the current line number, and whether or not the current line is from *style_file* or *change_file* or *mf_file*. This routine should be modified on systems whose standard text editor has special line-numbering conventions.

```
⟨ Print error location based on input buffer 30 ⟩ ≡
  begin if styling then print(`. (style file `)
  else if changing then print(`. (change file `) else print(`. (`);
  print_ln(`l.`, line : 1, `)`);
  if loc ≥ limit then l ← limit
  else l ← loc;
  for k ← 1 to l do print(xchr[buffer[k − 1]]); { print the characters already read }
  new_line;
  for k ← 1 to l do print(` `); { space out the next line }
  for k ← l + 1 to limit do print(xchr[buffer[k − 1]]); { print the part not yet read }
  end
```

This code is used in section 29.

31. The *jump_out* procedure just cuts across all active procedure levels and jumps out of the program. This is the only non-local **goto** statement in MFT. It is used when no recovery from a particular error has been provided.

Some Pascal compilers do not implement non-local **goto** statements. In such cases the code that appears at label *end_of_MFT* should be copied into the *jump_out* procedure, followed by a call to a system procedure that terminates the program.

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

```
⟨ Error handling procedures 29 ⟩ +≡
procedure jump_out;
  begin goto end_of_MFT;
  end;
```

32. Sometimes the program's behavior is far different from what it should be, and `MFT` prints an error message that is really for the `MFT` maintenance person, not the user. In such cases the program says *confusion*(´`indication␣of␣where␣we␣are`´).

define *confusion*(#) ≡ *fatal_error*(´`!␣This␣can´´t␣happen␣(´,#,´)´`)

33. An overflow stop occurs if `MFT`'s tables aren't large enough.

define *overflow*(#) ≡ *fatal_error*(´`!␣Sorry,␣´,#,´␣capacity␣exceeded´`)

34. Inserting the changes. Let's turn now to the low-level routine *get_line* that takes care of merging *change_file* into *mf_file*. The *get_line* procedure also updates the line numbers for error messages. (This routine was copied from `WEAVE`, but updated to include *styling*.)

```
⟨ Globals in the outer block 9 ⟩ +≡
line: integer;  { the number of the current line in the current file }
other_line: integer;  { the number of the current line in the input file that is not currently being read }
temp_line: integer;  { used when interchanging line with other_line }
limit: 0 .. buf_size;  { the last character position occupied in the buffer }
loc: 0 .. buf_size;  { the next character position to be read from the buffer }
input_has_ended: boolean;  { if true, there is no more input }
changing: boolean;  { if true, the current line is from change_file }
styling: boolean;  { if true, the current line is from style_file }
```

35. As we change *changing* from *true* to *false* and back again, we must remember to swap the values of *line* and *other_line* so that the *err_print* routine will be sure to report the correct line number.

```
define change_changing ≡ changing ← ¬changing; temp_line ← other_line; other_line ← line;
        line ← temp_line  { line ↔ other_line }
```

36. When *changing* is *false*, the next line of *change_file* is kept in *change_buffer*[0 .. *change_limit*], for purposes of comparison with the next line of *mf_file*. After the change file has been completely input, we set *change_limit* ← 0, so that no further matches will be made.

```
⟨ Globals in the outer block 9 ⟩ +≡
change_buffer: array [0 .. buf_size] of ASCII_code;
change_limit: 0 .. buf_size;  { the last position occupied in change_buffer }
```

37. Here's a simple function that checks if the two buffers are different.

```
function lines_dont_match: boolean;
  label exit;
  var k: 0 .. buf_size;  { index into the buffers }
  begin lines_dont_match ← true;
  if change_limit ≠ limit then return;
  if limit > 0 then
    for k ← 0 to limit − 1 do
      if change_buffer[k] ≠ buffer[k] then return;
  lines_dont_match ← false;
exit: end;
```

38. Procedure *prime_the_change_buffer* sets *change_buffer* in preparation for the next matching operation. Since blank lines in the change file are not used for matching, we have (*change_limit* = 0) ∧ ¬*changing* if and only if the change file is exhausted. This procedure is called only when *changing* is true; hence error messages will be reported correctly.

```
procedure prime_the_change_buffer;
  label continue, done, exit;
  var k: 0 .. buf_size;  { index into the buffers }
  begin change_limit ← 0;  { this value will be used if the change file ends }
  ⟨ Skip over comment lines in the change file; return if end of file 39 ⟩;
  ⟨ Skip to the next nonblank line; return if end of file 40 ⟩;
  ⟨ Move buffer and limit to change_buffer and change_limit 41 ⟩;
exit: end;
```

39. While looking for a line that begins with `@x` in the change file, we allow lines that begin with `@`, as long as they don't begin with `@y` or `@z` (which would probably indicate that the change file is fouled up).

```
⟨ Skip over comment lines in the change file; return if end of file 39 ⟩ ≡
  loop begin incr(line);
    if ¬input_ln(change_file) then return;
    if limit < 2 then goto continue;
    if buffer[0] ≠ "@" then goto continue;
    if (buffer[1] ≥ "X") ∧ (buffer[1] ≤ "Z") then buffer[1] ← buffer[1] + "z" − "Z";  { lowercasify }
    if buffer[1] = "x" then goto done;
    if (buffer[1] = "y")|(buffer[1] = "z") then
      begin loc ← 2; err_print('!␣Where␣is␣the␣matching␣@x?');
      end;
  continue: end;
done:
```

This code is used in section 38.

40. Here we are looking at lines following the `@x`.

```
⟨ Skip to the next nonblank line; return if end of file 40 ⟩ ≡
  repeat incr(line);
    if ¬input_ln(change_file) then
      begin err_print('!␣Change␣file␣ended␣after␣@x'); return;
      end;
  until limit > 0;
```

This code is used in section 38.

41.

```
⟨ Move buffer and limit to change_buffer and change_limit 41 ⟩ ≡
  begin change_limit ← limit;
  if limit > 0 then
    for k ← 0 to limit − 1 do change_buffer[k] ← buffer[k];
  end
```

This code is used in sections 38 and 42.

42. The following procedure is used to see if the next change entry should go into effect; it is called only when *changing* is false. The idea is to test whether or not the current contents of *buffer* matches the current contents of *change_buffer*. If not, there's nothing more to do; but if so, a change is called for: All of the text down to the `@y` is supposed to match. An error message is issued if any discrepancy is found. Then the procedure prepares to read the next line from *change_file*.

```
procedure check_change; { switches to change_file if the buffers match }
  label exit;
  var n: integer; { the number of discrepancies found }
    k: 0 .. buf_size; { index into the buffers }
  begin if lines_dont_match then return;
  n ← 0;
  loop begin change_changing; { now it's true }
    incr(line);
    if ¬input_ln(change_file) then
      begin err_print('!␣Change␣file␣ended␣before␣@y'); change_limit ← 0; change_changing;
          { false again }
      return;
      end;
    ⟨ If the current line starts with @y, report any discrepancies and return 43 ⟩;
    ⟨ Move buffer and limit to change_buffer and change_limit 41 ⟩;
    change_changing; { now it's false }
    incr(line);
    if ¬input_ln(mf_file) then
      begin err_print('!␣MF␣file␣ended␣during␣a␣change'); input_has_ended ← true; return;
      end;
    if lines_dont_match then incr(n);
    end;
exit: end;
```

43. ⟨ If the current line starts with `@y`, report any discrepancies and **return** 43 ⟩ ≡

```
if limit > 1 then
  if buffer[0] = "@" then
    begin if (buffer[1] ≥ "X") ∧ (buffer[1] ≤ "Z") then buffer[1] ← buffer[1] + "z" − "Z";
        { lowercasify }
    if (buffer[1] = "x")|(buffer[1] = "z") then
      begin loc ← 2; err_print('!␣Where␣is␣the␣matching␣@y?');
      end
    else if buffer[1] = "y" then
      begin if n > 0 then
        begin loc ← 2;
        err_print('!␣Hmm...␣', n : 1, '␣of␣the␣preceding␣lines␣failed␣to␣match');
        end;
      return;
      end;
    end
```

This code is used in section 42.

44. Here's what we do to get the input rolling.

```
⟨ Initialize the input system 44 ⟩ ≡
  begin open_input; line ← 0; other_line ← 0;
  changing ← true; prime_the_change_buffer; change_changing;
  styling ← true; limit ← 0; loc ← 1; buffer[0] ← "␣"; input_has_ended ← false;
  end
```

This code is used in section 112.

45. The *get_line* procedure is called when *loc* > *limit*; it puts the next line of merged input into the buffer and updates the other variables appropriately.

```
procedure get_line; { inputs the next line }
  label restart;
  begin restart: if styling then ⟨ Read from style_file and maybe turn off styling 47 ⟩;
  if ¬styling then
    begin if changing then ⟨ Read from change_file and maybe turn off changing 48 ⟩;
    if ¬changing then
      begin ⟨ Read from mf_file and maybe turn on changing 46 ⟩;
      if changing then goto restart;
      end;
    end;
  end;
```

46.

```
⟨ Read from mf_file and maybe turn on changing 46 ⟩ ≡
  begin incr(line);
  if ¬input_ln(mf_file) then input_has_ended ← true
  else if limit = change_limit then
      if buffer[0] = change_buffer[0] then
        if change_limit > 0 then check_change;
  end
```

This code is used in section 45.

47.

```
⟨ Read from style_file and maybe turn off styling 47 ⟩ ≡
  begin incr(line);
  if ¬input_ln(style_file) then
    begin styling ← false; line ← 0;
    end;
  end
```

This code is used in section 45.

48. ⟨ Read from *change_file* and maybe turn off *changing* 48 ⟩ ≡

```
begin incr(line);
if ¬input_ln(change_file) then
  begin err_print('!␣Change␣file␣ended␣without␣@z'); buffer[0] ← "@"; buffer[1] ← "z"; limit ← 2;
  end;
if limit > 1 then { check if the change has ended }
  if buffer[0] = "@" then
    begin if (buffer[1] ≥ "X") ∧ (buffer[1] ≤ "Z") then buffer[1] ← buffer[1] + "z" − "Z";
          { lowercasify }
    if (buffer[1] = "x")|(buffer[1] = "y") then
      begin loc ← 2; err_print('!␣Where␣is␣the␣matching␣@z?');
      end
    else if buffer[1] = "z" then
        begin prime_the_change_buffer; change_changing;
        end;
    end;
end
```

This code is used in section 45.

49. At the end of the program, we will tell the user if the change file had a line that didn't match any relevant line in *mf_file*.

⟨ Check that all changes have been read 49 ⟩ ≡

```
if change_limit ≠ 0 then { changing is false }
  begin for loc ← 0 to change_limit do buffer[loc] ← change_buffer[loc];
  limit ← change_limit; changing ← true; line ← other_line; loc ← change_limit;
  err_print('!␣Change␣file␣entry␣did␣not␣match');
  end
```

This code is used in section 112.

50. Data structures. `MFT` puts token names into the large *byte_mem* array, which is packed with eight-bit integers. Allocation is sequential, since names are never deleted.

An auxiliary array *byte_start* is used as a directory for *byte_mem*; the *link* and *ilk* arrays give further information about names. These auxiliary arrays consist of sixteen-bit items.

⟨ Types in the outer block 12 ⟩ +≡
  eight_bits = 0 .. 255; { unsigned one-byte quantity }
  sixteen_bits = 0 .. 65535; { unsigned two-byte quantity }

51. `MFT` has been designed to avoid the need for indices that are more than sixteen bits wide, so that it can be used on most computers.

⟨ Globals in the outer block 9 ⟩ +≡
byte_mem: **packed array** [0 .. *max_bytes*] **of** *ASCII_code*; { characters of names }
byte_start: **array** [0 .. *max_names*] **of** *sixteen_bits*; { directory into *byte_mem* }
link: **array** [0 .. *max_names*] **of** *sixteen_bits*; { hash table links }
ilk: **array** [0 .. *max_names*] **of** *sixteen_bits*; { type codes }

52. The names of tokens are found by computing a hash address h and then looking at strings of bytes signified by $hash[h]$, $link[hash[h]]$, $link[link[hash[h]]]$, ..., until either finding the desired name or encountering a zero.

A '*name_pointer*' variable, which signifies a name, is an index into *byte_start*. The actual sequence of characters in the name pointed to by p appears in positions $byte\_start[p]$ to $byte\_start[p+1]-1$, inclusive, of *byte_mem*.

We usually have $byte\_start[name\_ptr] = byte\_ptr$, which is the starting position for the next name to be stored in *byte_mem*.

define $length(\#) \equiv byte\_start[\# + 1] - byte\_start[\#]$ { the length of a name }

⟨ Types in the outer block 12 ⟩ +≡
  name_pointer = 0 .. *max_names*; { identifies a name }

53. ⟨ Globals in the outer block 9 ⟩ +≡
name_ptr: *name_pointer*; { first unused position in *byte_start* }
byte_ptr: 0 .. *max_bytes*; { first unused position in *byte_mem* }

54. ⟨ Set initial values 10 ⟩ +≡
  $byte\_start[0] \leftarrow 0$; $byte\_ptr \leftarrow 0$; $byte\_start[1] \leftarrow 0$; { this makes name 0 of length zero }
  $name\_ptr \leftarrow 1$;

55. The hash table described above is updated by the *lookup* procedure, which finds a given name and returns a pointer to its index in *byte_start*. The token is supposed to match character by character. If it was not already present, it is inserted into the table.

Because of the way `MFT`'s scanning mechanism works, it is most convenient to let *lookup* search for a token that is present in the *buffer* array. Two other global variables specify its position in the buffer: the first character is $buffer[id\_first]$, and the last is $buffer[id\_loc - 1]$.

⟨ Globals in the outer block 9 ⟩ +≡
id_first: 0 .. *buf_size*; { where the current token begins in the buffer }
id_loc: 0 .. *buf_size*; { just after the current token in the buffer }
hash: **array** [0 .. *hash_size*] **of** *sixteen_bits*; { heads of hash lists }

56. Initially all the hash lists are empty.

⟨ Local variables for initialization 14 ⟩ +≡
h: 0 .. *hash_size*; { index into hash-head array }

57. ⟨ Set initial values 10 ⟩ +≡
```
for h ← 0 to hash_size − 1 do hash[h] ← 0;
```

58. Here now is the main procedure for finding tokens.
```
function lookup: name_pointer; { finds current token }
  label found;
  var i: 0 .. buf_size; { index into buffer }
    h: 0 .. hash_size; { hash code }
    k: 0 .. max_bytes; { index into byte_mem }
    l: 0 .. buf_size; { length of the given token }
    p: name_pointer; { where the token is being sought }
  begin l ← id_loc − id_first; { compute the length }
  ⟨ Compute the hash code h 59 ⟩;
  ⟨ Compute the name location p 60 ⟩;
  if p = name_ptr then ⟨ Enter a new name into the table at position p 62 ⟩;
  lookup ← p;
  end;
```

59. A simple hash code is used: If the sequence of ASCII codes is $c_1 c_2 \ldots c_m$, its hash value will be

$$(2^{n-1}c_1 + 2^{n-2}c_2 + \cdots + c_n) \bmod hash\_size.$$

```
⟨ Compute the hash code h 59 ⟩ ≡
  h ← buffer[id_first]; i ← id_first + 1;
  while i < id_loc do
    begin h ← (h + h + buffer[i]) mod hash_size; incr(i);
    end
```
This code is used in section 58.

60. If the token is new, it will be placed in position $p = name\_ptr$, otherwise p will point to its existing location.
```
⟨ Compute the name location p 60 ⟩ ≡
  p ← hash[h];
  while p ≠ 0 do
    begin if length(p) = l then ⟨ Compare name p with current token, goto found if equal 61 ⟩;
    p ← link[p];
    end;
  p ← name_ptr; { the current token is new }
  link[p] ← hash[h]; hash[h] ← p; { insert p at beginning of hash list }
found:
```
This code is used in section 58.

61.
```
⟨ Compare name p with current token, goto found if equal 61 ⟩ ≡
  begin i ← id_first; k ← byte_start[p];
  while (i < id_loc) ∧ (buffer[i] = byte_mem[k]) do
    begin incr(i); incr(k);
    end;
  if i = id_loc then goto found; { all characters agree }
  end
```
This code is used in section 60.

62. When we begin the following segment of the program, $p = name\_ptr$.

```
⟨ Enter a new name into the table at position p 62 ⟩ ≡
  begin if byte_ptr + l > max_bytes then overflow('byte␣memory');
  if name_ptr + 1 > max_names then overflow('name');
  i ← id_first;  { get ready to move the token into byte_mem }
  while i < id_loc do
    begin byte_mem[byte_ptr] ← buffer[i]; incr(byte_ptr); incr(i);
    end;
  incr(name_ptr); byte_start[name_ptr] ← byte_ptr; ⟨ Assign the default value to ilk[p] 63 ⟩;
  end
```

This code is used in section 58.

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 *special_tag* = 32 { internal code for tags that take at most one subscript }
define *tag* = 33 { internal code for nonprimitive tokens }

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

This code is used in section 62.

64. We have to get METAFONT's primitives into the hash table, and the simplest way to do this is to insert them every time MFT is run.

A few macros permit us to do the initialization with a compact program. We use the fact that the longest primitive is `intersectiontimes`, which is 17 letters long.

```
define spr17(#) ≡ buffer[17] ← #; cur_tok ← lookup; ilk[cur_tok] ←
define spr16(#) ≡ buffer[16] ← #; spr17
define spr15(#) ≡ buffer[15] ← #; spr16
define spr14(#) ≡ buffer[14] ← #; spr15
define spr13(#) ≡ buffer[13] ← #; spr14
define spr12(#) ≡ buffer[12] ← #; spr13
define spr11(#) ≡ buffer[11] ← #; spr12
define spr10(#) ≡ buffer[10] ← #; spr11
define spr9(#) ≡ buffer[9] ← #; spr10
define spr8(#) ≡ buffer[8] ← #; spr9
define spr7(#) ≡ buffer[7] ← #; spr8
define spr6(#) ≡ buffer[6] ← #; spr7
define spr5(#) ≡ buffer[5] ← #; spr6
define spr4(#) ≡ buffer[4] ← #; spr5
define spr3(#) ≡ buffer[3] ← #; spr4
define spr2(#) ≡ buffer[2] ← #; spr3
define spr1(#) ≡ buffer[1] ← #; spr2
define pr1 ≡ id_first ← 17; spr17
define pr2 ≡ id_first ← 16; spr16
define pr3 ≡ id_first ← 15; spr15
define pr4 ≡ id_first ← 14; spr14
define pr5 ≡ id_first ← 13; spr13
define pr6 ≡ id_first ← 12; spr12
define pr7 ≡ id_first ← 11; spr11
define pr8 ≡ id_first ← 10; spr10
define pr9 ≡ id_first ← 9; spr9
define pr10 ≡ id_first ← 8; spr8
define pr11 ≡ id_first ← 7; spr7
define pr12 ≡ id_first ← 6; spr6
define pr13 ≡ id_first ← 5; spr5
define pr14 ≡ id_first ← 4; spr4
define pr15 ≡ id_first ← 3; spr3
define pr16 ≡ id_first ← 2; spr2
define pr17 ≡ id_first ← 1; spr1
```

65. 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);
  pr7("d")("i")("s")("p")("l")("a")("y")(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);
  pr8("i")("n")("w")("i")("n")("d")("o")("w")(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("o")("p")("e")("n")("w")("i")("n")("d")("o")("w")(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, and 71.

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*);
pr4(`"g"`)(`"o"`)(`"o"`)(`"d"`)(*special_tag*);
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);
pr11("t")("o")("t")("a")("l")("w")("e")("i")("g")("h")("t")(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);
pr12("t")("r")("a")("c")("i")("n")("g")("e")("d")("g")("e")("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);
```

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")("f")("a")("m")(*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*);
pr4("h")("p")("p")("p")(*internal*);
pr4("v")("p")("p")("p")(*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*);
pr9("s")("m")("o")("o")("t")("h")("i")("n")("g")(*internal*);
pr12("a")("u")("t")("o")("r")("o")("u")("n")("d")("i")("n")("g")(*internal*);
pr11("g")("r")("a")("n")("u")("l")("a")("r")("i")("t")("y")(*internal*);
pr6("f")("i")("l")("l")("i")("n")(*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);
pr10("w")("i")("t")("h")("w")("e")("i")("g")("h")("t")(command);
pr8("d")("r")("o")("p")("p")("i")("n")("g")(command);
pr7("k")("e")("e")("p")("i")("n")("g")(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);
pr10("n")("u")("m")("s")("p")("e")("c")("i")("a")("l")(command);
pr1("%")(comment);
pr2("%")("%")(verbatim);
pr3("%")("%")("%")(set_format);
pr4("%")("%")("%")("%")(mft_comment);
pr1("#")(sharp);
```

72. We also want to store a few other strings of characters that are used in MFT's translation to TEX code.

```
define ttr1(#) ≡ byte_mem[byte_ptr − 1] ← #; cur_tok ← name_ptr; incr(name_ptr);
        byte_start[name_ptr] ← byte_ptr
define ttr2(#) ≡ byte_mem[byte_ptr − 2] ← #; ttr1
define ttr3(#) ≡ byte_mem[byte_ptr − 3] ← #; ttr2
define ttr4(#) ≡ byte_mem[byte_ptr − 4] ← #; ttr3
define ttr5(#) ≡ byte_mem[byte_ptr − 5] ← #; ttr4
define tr1 ≡ incr(byte_ptr); ttr1
define tr2 ≡ byte_ptr ← byte_ptr + 2; ttr2
define tr3 ≡ byte_ptr ← byte_ptr + 3; ttr3
define tr4 ≡ byte_ptr ← byte_ptr + 4; ttr4
define tr5 ≡ byte_ptr ← byte_ptr + 5; ttr5
```

⟨ Globals in the outer block 9 ⟩ +≡

```
translation: array [ASCII_code] of name_pointer;
i: ASCII_code; { index into translation }
```

73. ⟨ Store all the translations 73 ⟩ ≡

```
for i ← 0 to 255 do translation[i] ← 0;
tr2("\")("$"); translation["$"] ← cur_tok;
tr2("\")("#"); translation["#"] ← cur_tok;
tr2("\")("&"); translation["&"] ← cur_tok;
tr2("\")("{"); translation["{"] ← cur_tok;
tr2("\")("}"); translation["}"] ← cur_tok;
tr2("\")("_"); translation["_"] ← cur_tok;
tr2("\")("%"); translation["%"] ← cur_tok;
tr4("\")("B")("S")("␣"); translation["\"] ← cur_tok;
tr4("\")("H")("A")("␣"); translation["^"] ← cur_tok;
tr4("\")("T")("I")("␣"); translation["~"] ← cur_tok;
tr5("\")("a")("s")("t")("␣"); translation["*"] ← cur_tok;
tr4("\")("A")("M")("␣"); tr_amp ← cur_tok;
tr4("\")("B")("L")("␣"); tr_skip ← cur_tok;
tr4("\")("S")("H")("␣"); tr_sharp ← cur_tok;
tr4("\")("P")("S")("␣"); tr_ps ← cur_tok;
tr4("\")("l")("e")("␣"); tr_le ← cur_tok;
tr4("\")("g")("e")("␣"); tr_ge ← cur_tok;
tr4("\")("n")("e")("␣"); tr_ne ← cur_tok;
tr5("\")("q")("u")("a")("d"); tr_quad ← cur_tok;
```

This code is used in section 112.

74. ⟨ Globals in the outer block 9 ⟩ +≡

```
tr_le, tr_ge, tr_ne, tr_amp, tr_sharp, tr_skip, tr_ps, tr_quad: name_pointer; { special translations }
```

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*.

⟨ 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* }

76. ⟨ Set initial values 10 ⟩ +≡
cur_type ← *end_of_line*; *cur_tok* ← 0;

77. Two global state variables affect the behavior of *get_next*: A space will be considered significant when *start_of_line* is *true*, and the buffer will be considered devoid of information when *empty_buffer* is *true*.

⟨ Globals in the outer block 9 ⟩ +≡
start_of_line: *boolean*; { has the current line had nothing but spaces so far? }
empty_buffer: *boolean*; { is it time to input a new line? }

78. 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, expect 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 METAFONT.

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, 6, 7, 8 { characters that make length-one tokens only }
define *letter_class* = 9 { letters and the underline character }
define *left_bracket_class* = 17 { '[' }
define *right_bracket_class* = 18 { ']' }
define *invalid_class* = 20 { bad character in the input }
define *end_line_class* = 21 { end of an input line (MFT only) }
define *max_class* = 21 { the largest class number }

⟨ Globals in the outer block 9 ⟩ +≡
char_class: **array** [*ASCII_code*] **of** 0 .. *max_class*; { the class numbers }

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*] ← *invalid_class*;
char_class[*carriage_return*] ← *end_line_class*;
for *i* ← 127 **to** 255 **do** *char_class*[*i*] ← *invalid_class*;

80. And now we're ready to take the plunge into *get_next* itself.

define *switch* = 25 { a label in *get_next* }
define *pass_digits* = 85 { another }
define *pass_fraction* = 86 { and still another, although **goto** is considered harmful }

procedure *get_next*; { sets *cur_type* and *cur_tok* to next token }
label *switch*, *pass_digits*, *pass_fraction*, *done*, *found*, *exit*;
var *c*: *ASCII_code*; { the current character in the buffer }
class: *ASCII_code*; { its class number }
begin *prev_type* ← *cur_type*; *prev_tok* ← *cur_tok*;
if *empty_buffer* **then** ⟨ Bring in a new line of input; **return** if the file has ended 85 ⟩;
switch: *c* ← *buffer*[*loc*]; *id_first* ← *loc*; *incr*(*loc*); *class* ← *char_class*[*c*]; ⟨ 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 ⟩;
found: *id_loc* ← *loc*; *cur_tok* ← *lookup*; *cur_type* ← *ilk*[*cur_tok*];
exit: **end**;

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 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.

82. ⟨ Get a string token and **return** 82 ⟩ ≡

```
loop begin if buffer[loc] = """" then
    begin incr(loc); emit(string_token);
    end;
  if loc = limit then ⟨ Decry the missing string delimiter and goto switch 83 ⟩;
  incr(loc);
  end
```

This code is used in section 81.

83. ⟨ Decry the missing string delimiter and **goto** *switch* 83 ⟩ ≡

```
begin err_print('!␣Incomplete␣string␣will␣be␣ignored'); goto switch;
end
```

This code is used in section 82.

84. ⟨ Decry the invalid character and **goto** *switch* 84 ⟩ ≡

```
begin err_print('!␣Invalid␣character␣will␣be␣ignored'); goto switch;
end
```

This code is used in section 81.

85. ⟨ Bring in a new line of input; **return** if the file has ended 85 ⟩ ≡

```
begin get_line;
if input_has_ended then emit(end_of_file);
buffer[limit] ← carriage_return; loc ← 0; start_of_line ← true; empty_buffer ← false;
end
```

This code is used in section 80.

86. Low-level output routines. The TEX output is supposed to appear in lines at most *line_length* characters long, so we place it into an output buffer. During the output process, *out_line* will hold the current line number of the line about to be output.

```
⟨ Globals in the outer block 9 ⟩ +≡
out_buf: array [0 .. line_length] of ASCII_code;  { assembled characters }
out_ptr: 0 .. line_length;  { number of characters in out_buf }
out_line: integer;  { coordinates of next line to be output }
```

87. The *flush_buffer* routine empties the buffer up to a given breakpoint, and moves any remaining characters to the beginning of the next line. If the *per_cent* parameter is *true*, a `"%"` is appended to the line that is being output; in this case the breakpoint b should be strictly less than *line_length*. If the *per_cent* parameter is *false*, trailing blanks are suppressed. The characters emptied from the buffer form a new line of output.

```
procedure flush_buffer(b : eight_bits; per_cent : boolean);  { outputs out_buf[1 .. b], where b ≤ out_ptr }
  label done;
  var j, k: 0 .. line_length;
  begin j ← b;
  if ¬per_cent then  { remove trailing blanks }
    loop begin if j = 0 then goto done;
      if out_buf[j] ≠ "␣" then goto done;
      decr(j);
      end;
done: for k ← 1 to j do write(tex_file, xchr[out_buf[k]]);
  if per_cent then write(tex_file, xchr["%"]);
  write_ln(tex_file); incr(out_line);
  if b < out_ptr then
    for k ← b + 1 to out_ptr do out_buf[k − b] ← out_buf[k];
  out_ptr ← out_ptr − b;
  end;
```

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`'.

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

89. When we wish to append the character c to the output buffer, we write '*out*(c)'; this will cause the buffer to be emptied if it was already full. Similarly, '*out2*(c_1)(c_2)' appends a pair of characters. A line break will occur at a space or after a single-nonletter TEX control sequence.

```
  define oot(#) ≡
      if out_ptr = line_length then break_out;
      incr(out_ptr); out_buf[out_ptr] ← #;
  define oot1(#) ≡ oot(#) end
  define oot2(#) ≡ oot(#) oot1
  define oot3(#) ≡ oot(#) oot2
  define oot4(#) ≡ oot(#) oot3
  define oot5(#) ≡ oot(#) oot4
  define out ≡ begin oot1
  define out2 ≡ begin oot2
  define out3 ≡ begin oot3
  define out4 ≡ begin oot4
  define out5 ≡ begin oot5
```

90. The *break_out* routine is called just before the output buffer is about to overflow. To make this routine a little faster, we initialize position 0 of the output buffer to '\'; this character isn't really output.

```
⟨ Set initial values 10 ⟩ +≡
  out_buf[0] ← "\";
```

91. A long line is broken at a blank space or just before a backslash that isn't preceded by another backslash. In the latter case, a `"%"` is output at the break. (This policy has a known bug, in the rare situation that the backslash was in a string constant that's being output "verbatim.")

```
procedure break_out; { finds a way to break the output line }
  label exit;
  var k: 0 .. line_length; { index into out_buf }
    d: ASCII_code; { character from the buffer }
  begin k ← out_ptr;
  loop begin if k = 0 then ⟨ Print warning message, break the line, return 92 ⟩;
    d ← out_buf[k];
    if d = "␣" then
      begin flush_buffer(k, false); return;
      end;
    if (d = "\") ∧ (out_buf[k − 1] ≠ "\") then { in this case k > 1 }
      begin flush_buffer(k − 1, true); return;
      end;
    decr(k);
    end;
exit: end;
```

92. We get to this module only in unusual cases that the entire output line consists of a string of backslashes followed by a string of nonblank non-backslashes. In such cases it is almost always safe to break the line by putting a `"%"` just before the last character.

```
⟨ Print warning message, break the line, return 92 ⟩ ≡
  begin print_nl('!␣Line␣had␣to␣be␣broken␣(output␣l.', out_line : 1); print_ln('):');
  for k ← 1 to out_ptr − 1 do print(xchr[out_buf[k]]);
  new_line; mark_harmless; flush_buffer(out_ptr − 1, true); return;
  end
```

This code is used in section 91.

93. To output a string of bytes from *byte_mem*, we call *out_str*.

```
procedure out_str(p : name_pointer); { outputs a string }
  var k: 0 .. max_bytes; { index into byte_mem }
  begin for k ← byte_start[p] to byte_start[p + 1] − 1 do out(byte_mem[k]);
  end;
```

94. The *out_name* subroutine is used to output a symbolic token. Unusual characters are translated into forms that won't screw up.

```
procedure out_name(p : name_pointer);  { outputs a name }
  var k: 0 .. max_bytes;  { index into byte_mem }
    t: name_pointer;  { translation of character being output, if any }
  begin for k ← byte_start[p] to byte_start[p + 1] − 1 do
    begin t ← translation[byte_mem[k]];
    if t = 0 then out(byte_mem[k])
    else out_str(t);
    end;
  end;
```

95. We often want to output a name after calling a numeric macro (e.g., '`\1{foo}`').

```
procedure out_mac_and_name(n : ASCII_code; p : name_pointer);
  begin out("\"); out(n);
  if length(p) = 1 then out_name(p)
  else begin out("{"); out_name(p); out("}");
    end;
  end;
```

96. Here's a routine that simply copies from the input buffer to the output buffer.

```
procedure copy(first_loc : integer);  { output buffer[first_loc .. loc − 1] }
  var k: 0 .. buf_size;  { buffer location being copied }
  begin for k ← first_loc to loc − 1 do out(buffer[k]);
  end;
```

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 ⟩;
    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;
```

98. ⟨ Do special actions at the start of a line 98 ⟩ ≡

```
if cur_type ≥ min_action_type then
  begin out("$"); start_of_line ← false;
  case cur_type of
  endit: out2("\")("!");
  binary, abinary, bbinary, ampersand, pyth_sub: out2("{")("}");
  othercases do_nothing
  endcases;
  end
else if cur_type = end_of_line then
    begin out_str(tr_skip); goto restart;
    end
  else if cur_type = mft_comment then goto restart
```

This code is used in section 97.

99. Let's start with some of the easier translations, so that the harder ones will also be easy when we get to them. A string like "cat" comes out '\7"cat"'.

⟨ Translate a string token 99 ⟩ ≡

```
begin out2("\")("7"); copy(cur_tok);
end
```

This code is used in section 97.

100. Similarly, the translation of '`sqrt`' is '`\1{sqrt}`'.

```
⟨ Cases that translate primitive tokens 100 ⟩ ≡
op: out_mac_and_name("1", cur_tok);
command: out_mac_and_name("2", cur_tok);
type_name: if prev_type = command then out_mac_and_name("1", cur_tok)
  else out_mac_and_name("2", cur_tok);
endit: out_mac_and_name("3", cur_tok);
bbinary: out_mac_and_name("4", cur_tok);
bold: out_mac_and_name("5", cur_tok);
binary: out_mac_and_name("6", cur_tok);
path_join: out_mac_and_name("8", cur_tok);
colon: out_mac_and_name("?", cur_tok);
```

See also sections 101, 102, and 103.

This code is used in section 97.

101. Here are a few more easy cases.

```
⟨ Cases that translate primitive tokens 100 ⟩ +≡
as_is, sharp, abinary: out_name(cur_tok);
double_back: out2("\")(";");
semicolon: begin out_name(cur_tok); get_next;
  if cur_type ≠ end_of_line then
    if cur_type ≠ endit then out2("\")("␣");
  goto reswitch;
  end;
```

102. Some of the primitives have a fixed output (independent of *cur_tok*):

```
⟨ Cases that translate primitive tokens 100 ⟩ +≡
backslash: out_str(translation["\"]);
pyth_sub: out_str(tr_ps);
less_or_equal: out_str(tr_le);
greater_or_equal: out_str(tr_ge);
not_equal: out_str(tr_ne);
ampersand: out_str(tr_amp);
```

103. The remaining primitive is slightly special.

```
⟨ Cases that translate primitive tokens 100 ⟩ +≡
input_command: begin out_mac_and_name("2", cur_tok); out5("\")("h")("b")("o")("x");
  ⟨ Scan the file name and output it in typewriter type 104 ⟩;
  end;
```

104. File names have different formats on different computers, so we don't scan them with *get_next*. Here we use a rule that probably covers most cases satisfactorily: We ignore leading blanks, then consider the file name to consist of all subsequent characters up to the first blank, semicolon, comment, or end-of-line. (A *carriage_return* appears at the end of the line.)

```
⟨ Scan the file name and output it in typewriter type 104 ⟩ ≡
  while buffer[loc] = "␣" do incr(loc);
  out5("{")("\")("t")("t")("␣");
  while (buffer[loc] ≠ "␣") ∧ (buffer[loc] ≠ "%") ∧ (buffer[loc] ≠ ";") ∧ (loc < limit) do
    begin out(buffer[loc]); incr(loc);
    end;
  out("}")
```

This code is used in section 103.

105.

```
⟨ Translate a numeric token or a fraction 105 ⟩ ≡
  if buffer[loc] = "/" then
    if char_class[buffer[loc + 1]] = digit_class then { it's a fraction }
      begin out5("\")("f")("r")("a")("c"); copy(cur_tok); get_next; out2("/")("{"); get_next;
      copy(cur_tok); out("}");
      end
    else copy(cur_tok)
  else copy(cur_tok)
```

This code is used in section 97.

106.

```
⟨ Translate a tag and possible subscript 106 ⟩ ≡
  begin if length(cur_tok) = 1 then out_name(cur_tok)
  else out_mac_and_name("\", cur_tok);
  get_next;
  if byte_mem[byte_start[prev_tok]] = "´" then goto reswitch;
  case prev_type of
  internal: begin if (cur_type = numeric_token)|(cur_type ≥ min_suffix) then out2("\")(",");
    goto reswitch;
    end;
  special_tag: if cur_type < min_suffix then goto reswitch
    else begin out("."); cur_type ← internal; goto reswitch;
      end;
  tag: begin if cur_type = tag then
      if byte_mem[byte_start[cur_tok]] = "´" then goto reswitch;
          { a sequence of primes goes on the main line }
    if (cur_type = numeric_token)|(cur_type ≥ min_suffix) then ⟨ Translate a subscript 107 ⟩
    else if cur_type = sharp then out_str(tr_sharp)
      else goto reswitch;
    end;
  end; { there are no other cases }
  end
```

This code is used in section 97.

107. ⟨ Translate a subscript 107 ⟩ ≡

```
begin out2("_")("{");
loop begin if cur_type ≥ min_suffix then out_name(cur_tok)
  else copy(cur_tok);
  if prev_type = special_tag then
    begin get_next; goto done;
    end;
  get_next;
  if cur_type < min_suffix then
    if cur_type ≠ numeric_token then goto done;
  if cur_type = prev_type then
    if cur_type = numeric_token then out2("\")(",")
    else if char_class[byte_mem[byte_start[cur_tok]]] = char_class[byte_mem[byte_start[prev_tok]]] then
        if byte_mem[byte_start[prev_tok]] ≠ "." then out(".")
        else out2("\")(",");
  end;
done: out("}"); goto reswitch;
  end
```

This code is used in section 106.

108. The tricky thing about comments is that they might contain `|...|`. We scan ahead for this, and replace the second '`|`' by a *carriage_return*.

⟨ Translate a comment and **goto** *restart*, unless there's a `|...|` segment 108 ⟩ ≡

```
begin if cur_type = comment then out2("\")("9");
id_first ← loc;
while (loc < limit) ∧ (buffer[loc] ≠ "|") do incr(loc);
copy(id_first);
if loc < limit then
  begin start_of_line ← true; incr(loc); k ← loc;
  while (k < limit) ∧ (buffer[k] ≠ "|") do incr(k);
  buffer[k] ← carriage_return;
  end
else begin if out_buf[out_ptr] = "\" then out("␣");
  out4("\")("p")("a")("r"); goto restart;
  end;
end
```

This code is used in section 97.

109. ⟨ Copy the rest of the current input line to the output, then **goto** *restart* 109 ⟩ ≡

```
begin id_first ← loc; loc ← limit; copy(id_first);
if out_ptr = 0 then
  begin out_ptr ← 1; out_buf[1] ← "␣";
  end;
goto restart;
end
```

This code is used in section 97.

110. ⟨ Wind up a line of translation and **goto** *restart*, or finish a `|...|` segment and **goto** *reswitch* 110 ⟩ ≡

```
begin out("$");
if (loc < limit) ∧ (cur_type = end_of_line) then
  begin cur_type ← recomment; goto reswitch;
  end
else begin out4("\")("p")("a")("r"); goto restart;
  end;
end
```

This code is used in section 97.

111. ⟨ Change the translation format of tokens, and **goto** *restart* or *reswitch* 111 ⟩ ≡

```
begin start_of_line ← false; get_next; t ← cur_type;
while cur_type ≥ min_symbolic_token do
  begin get_next;
  if cur_type ≥ min_symbolic_token then ilk[cur_tok] ← t;
  end;
if cur_type ≠ end_of_line then
  if cur_type ≠ mft_comment then
    begin err_print('!␣Only␣symbolic␣tokens␣should␣appear␣after␣%%%'); goto reswitch;
    end;
empty_buffer ← true; goto restart;
end
```

This code is used in section 97.

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

```
  begin initialize; { beginning of the main program }
  print_ln(banner); { print a "banner line" }
  ⟨ 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 ⟩;
end_of_MFT: { here files should be closed if the operating system requires it }
  ⟨ Print the job history 113 ⟩;
  end.
```

113. Some implementations may wish to pass the *history* value to the operating system so that it can be used to govern whether or not other programs are started. Here we simply report the history to the user.

```
⟨ Print the job history 113 ⟩ ≡
  case history of
  spotless: print_nl('(No␣errors␣were␣found.)');
  harmless_message: print_nl('(Did␣you␣see␣the␣warning␣message␣above?)');
  error_message: print_nl('(Pardon␣me,␣but␣I␣think␣I␣spotted␣something␣wrong.)');
  fatal_message: print_nl('(That␣was␣a␣fatal␣error,␣my␣friend.)');
  end { there are no other cases }
```

This code is used in section 112.

114. System-dependent changes. This module should be replaced, if necessary, by changes to the program that are necessary to make `MFT` work at a particular installation. It is usually best to design your change file so that all changes to previous modules preserve the module numbering; then everybody's version will be consistent with the printed program. More extensive changes, which introduce new modules, can be inserted here; then only the index itself will get a new module number.

115. Index.