The GFtoDVI processor

(Version 3.0, 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. 'TEX' is a trademark of the American Mathematical Society. 'METAFONT' is a trademark of Addison-Wesley Publishing Company.

1. Introduction. The GFtoDVI utility program reads binary generic font ("GF") files that are produced by font compilers such as METAFONT, and converts them into device-independent ("DVI") files that can be printed to give annotated hardcopy proofs of the character shapes. The annotations are specified by the comparatively simple conventions of plain METAFONT; i.e., there are mechanisms for labeling chosen points and for superimposing horizontal or vertical rules on the enlarged character shapes.

The purpose of GFtoDVI is simply to make proof copies; it does not exhaustively test the validity of a GF file, nor do its algorithms much resemble the algorithms that are typically used to prepare font descriptions for commercial typesetting equipment. Another program, GFtype, is available for validity checking; GFtype also serves as a model of programs that convert fonts from GF format to some other coding scheme.

The *banner* string defined here should be changed whenever GFtoDVI gets modified.

define *banner* ≡ ´This␣is␣GFtoDVI,␣Version␣3.0´ { printed when the program starts }

2. This program is written in standard Pascal, except where it is necessary to use extensions; for example, GFtoDVI must read files whose names are dynamically specified, and such a task would be impossible in pure Pascal. All places where nonstandard constructions are used have been listed in the index under "system dependencies."

Another exception to standard Pascal occurs in the use of default branches in **case** statements; the conventions of TANGLE, WEAVE, etc., have been followed.

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*

3. The main input and output files are not mentioned in the program header, because their external names will be determined at run time (e.g., by interpreting the command line that invokes this program). Error messages and other remarks are written on the *output* file, which the user may choose to assign to the terminal if the system permits it.

The term *print* is used instead of *write* when this program writes on the *output* file, so that all such output can be easily deflected.

define *print*(#) ≡ *write*(#)
define *print_ln*(#) ≡ *write_ln*(#)
define *print_nl*(#) ≡ **begin** *write_ln*; *write*(#); **end**

```
program GF_to_DVI(output);
  label ⟨ Labels in the outer block 4 ⟩
  const ⟨ Constants in the outer block 5 ⟩
  type ⟨ Types in the outer block 9 ⟩
  var ⟨ Globals in the outer block 12 ⟩
  procedure initialize; { this procedure gets things started properly }
    var i, j, m, n: integer; { loop indices for initializations }
    begin print_ln(banner);
    ⟨ Set initial values 13 ⟩
    end;
```

4. If the program has to stop prematurely, it goes to the '*final_end*'.

define *final_end* = 9999 { label for the end of it all }

```
⟨ Labels in the outer block 4 ⟩ ≡
  final_end;
```

This code is used in section 3.

5. The following parameters can be changed at compile time to extend or reduce GFtoDVI's capacity.

⟨ Constants in the outer block 5 ⟩ ≡
```
max_labels = 2000; { maximum number of labels and dots and rules per character }
pool_size = 10000; { maximum total length of labels and other strings }
max_strings = 1100; { maximum number of labels and other strings }
terminal_line_length = 150;
    { maximum number of characters input in a single line of input from the terminal }
file_name_size = 50; { a file name shouldn't be longer than this }
font_mem_size = 2000; { space for font metric data }
dvi_buf_size = 800; { size of the output buffer; must be a multiple of 8 }
widest_row = 8192; { maximum number of pixels per row }
lig_lookahead = 20; { size of stack used when inserting ligature characters }
```
This code is used in section 3.

6. Labels are given symbolic names by the following definitions, so that occasional **goto** statements will be meaningful. We insert the label '*exit*:' just before the '**end**' of a procedure in which we have used the '**return**' statement defined below; 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*'.

Incidentally, this program never declares a label that isn't actually used, because some fussy Pascal compilers will complain about redundant labels.

```
define exit = 10 { go here to leave a procedure }
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 done1 = 31 { like done, when there is more than one loop }
define found = 40 { go here when you've found it }
define not_found = 45 { go here when you've found nothing }
```

7. 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 }
format loop ≡ xclause { WEB's xclause acts like 'while true do' }
define do_nothing ≡ { empty statement }
define return ≡ goto exit { terminate a procedure call }
format return ≡ nil { WEB will henceforth say return instead of return }
```

8. If the GF file is badly malformed, the whole process must be aborted; GFtoDVI will give up, after issuing an error message about the symptoms that were noticed.

Such errors might be discovered inside of subroutines inside of subroutines, so a procedure called *jump_out* has been introduced. This procedure, which simply transfers control to the label *final_end* at the end of the program, contains the only non-local **goto** statement in GFtoDVI.

```
define abort(#) ≡ begin print('␣', #); jump_out; end
define bad_gf(#) ≡ abort('Bad␣GF␣file:␣', #, '!␣(at␣byte␣', cur_loc − 1 : 1, ')')
procedure jump_out;
  begin goto final_end;
  end;
```

9. As in TEX and METAFONT, this program deals with numeric quantities that are integer multiples of 2^{16}, and calls them *scaled*.

define *unity* ≡ *´200000* { *scaled* representation of 1.0 }

⟨ Types in the outer block 9 ⟩ ≡
scaled = *integer*; { fixed-point numbers }

See also sections 10, 11, 45, 52, 70, 79, 104, and 139.

This code is used in section 3.

10. The character set. Like all programs written with the WEB system, GFtoDVI can be used with any character set. But it uses ASCII code internally, because the programming for portable input-output is easier when a fixed internal code is used. Furthermore, both GF and DVI files use ASCII code for file names and certain other strings. The next few sections of GFtoDVI have therefore been copied from the analogous ones in the WEB system routines.

⟨ Types in the outer block 9 ⟩ +≡
ASCII_code = 0 . . 255; { eight-bit numbers, a subrange of the integers }

11. 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. So we shall assume that the Pascal system being used for GFtoDVI has a character set containing at least the standard visible ASCII characters ("!" through "~"). If additional characters are present, GFtoDVI can be configured to work with them too.

Some Pascal compilers use the original name *char* for the data type associated with the characters in text files, 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 in the output file. 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 9 ⟩ +≡
text_file = **packed file of** *text_char*;

12. The GFtoDVI 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 12 ⟩ ≡
xord: **array** [*text_char*] **of** *ASCII_code*; { specifies conversion of input characters }
xchr: **array** [*ASCII_code*] **of** *text_char*; { specifies conversion of output characters }

See also sections 16, 18, 37, 46, 48, 49, 53, 71, 76, 80, 86, 87, 93, 96, 102, 105, 117, 127, 134, 140, 149, 155, 158, 160, 166, 168, 174, 182, 183, 207, 211, 212, and 220.

This code is used in section 3.

13. Under our assumption that the visible characters of standard ASCII are all present, the following assignment statements initialize the *xchr* array properly, without needing any system-dependent changes.

```
⟨ Set initial values 13 ⟩ ≡
  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] ← ´~´;
```

See also sections 14, 15, 54, 97, 103, 106, 118, 126, and 142.

This code is used in section 3.

14. Here now is the system-dependent part of the character set. If `GFtoDVI` is being implemented on a garden-variety Pascal for which only standard ASCII codes will appear in the input and output files, you don't need to make any changes here. But if you have, for example, an extended character set like the one in Appendix C of *The TEXbook*, the first line of code in this module should be changed to

$$\textbf{for } i \leftarrow 0 \textbf{ to } \acute{}37 \textbf{ do } xchr[i] \leftarrow chr(i);$$

`WEB`'s character set is essentially identical to TEX's.

```
⟨ Set initial values 13 ⟩ +≡
  for i ← 0 to ´37 do xchr[i] ← ´?´;
  for i ← ´177 to ´377 do xchr[i] ← ´?´;
```

15. The following system-independent code makes the *xord* array contain a suitable inverse to the information in *xchr*.

```
⟨ Set initial values 13 ⟩ +≡
  for i ← first_text_char to last_text_char do xord[chr(i)] ← "␣";
  for i ← 1 to ´377 do xord[xchr[i]] ← i;
  xord[´?´] ← "?";
```

16. The *input_ln* routine waits for the user to type a line at his or her terminal; then it puts ASCII-code equivalents for the characters on that line into the *buffer* array. The *term_in* file is used for terminal input.

Since the terminal is being used for both input and output, some systems need a special routine to make sure that the user can see a prompt message before waiting for input based on that message. (Otherwise the message may just be sitting in a hidden buffer somewhere, and the user will have no idea what the program is waiting for.) We shall call a system-dependent subroutine *update_terminal* in order to avoid this problem.

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

```
⟨ Globals in the outer block 12 ⟩ +≡
buffer: array [0 .. terminal_line_length] of 0 .. 255;
term_in: text_file;  { the terminal, considered as an input file }
```

17. A global variable *line_length* records the first buffer position after the line just read.

```
procedure input_ln;  { inputs a line from the terminal }
  begin update_terminal; reset(term_in);
  if eoln(term_in) then read_ln(term_in);
  line_length ← 0;
  while (line_length < terminal_line_length) ∧ ¬eoln(term_in) do
    begin buffer[line_length] ← xord[term_in↑]; incr(line_length); get(term_in);
    end;
  end;
```

18. The global variable *buf_ptr* is used while scanning each line of input; it points to the first unread character in *buffer*.

```
⟨ Globals in the outer block 12 ⟩ +≡
buf_ptr: 0 .. terminal_line_length;  { the number of characters read }
line_length: 0 .. terminal_line_length;  { end of line read by input_ln }
```

19. Device-independent file format. Before we get into the details of GFtoDVI, we need to know exactly what DVI files are. The form of such files was designed by David R. Fuchs in 1979. Almost any reasonable typesetting device can be driven by a program that takes DVI files as input, and dozens of such DVI-to-whatever programs have been written. Thus, it is possible to print the output of document compilers like TEX on many different kinds of equipment. (The following material has been copied almost verbatim from the program for TEX.)

A DVI file is a stream of 8-bit bytes, which may be regarded as a series of commands in a machine-like language. The first byte of each command is the operation code, and this code is followed by zero or more bytes that provide parameters to the command. The parameters themselves may consist of several consecutive bytes; for example, the '*set_rule*' command has two parameters, each of which is four bytes long. Parameters are usually regarded as nonnegative integers; but four-byte-long parameters, and shorter parameters that denote distances, can be either positive or negative. Such parameters are given in two's complement notation. For example, a two-byte-long distance parameter has a value between -2^{15} and $2^{15}-1$.

Incidentally, when two or more 8-bit bytes are combined to form an integer of 16 or more bits, the most significant bytes appear first in the file. This is called BigEndian order.

A DVI file consists of a "preamble," followed by a sequence of one or more "pages," followed by a "postamble." The preamble is simply a *pre* command, with its parameters that define the dimensions used in the file; this must come first. Each "page" consists of a *bop* command, followed by any number of other commands that tell where characters are to be placed on a physical page, followed by an *eop* command. The pages appear in the order that they were generated, not in any particular numerical order. If we ignore *nop* commands and *fnt_def* commands (which are allowed between any two commands in the file), each *eop* command is immediately followed by a *bop* command, or by a *post* command; in the latter case, there are no more pages in the file, and the remaining bytes form the postamble. Further details about the postamble will be explained later.

Some parameters in DVI commands are "pointers." These are four-byte quantities that give the location number of some other byte in the file; the first byte is number 0, then comes number 1, and so on. For example, one of the parameters of a *bop* command points to the previous *bop*; this makes it feasible to read the pages in backwards order, in case the results are being directed to a device that stacks its output face up. Suppose the preamble of a DVI file occupies bytes 0 to 99. Now if the first page occupies bytes 100 to 999, say, and if the second page occupies bytes 1000 to 1999, then the *bop* that starts in byte 1000 points to 100 and the *bop* that starts in byte 2000 points to 1000. (The very first *bop*, i.e., the one that starts in byte 100, has a pointer of -1.)

20. The DVI format is intended to be both compact and easily interpreted by a machine. Compactness is achieved by making most of the information implicit instead of explicit. When a DVI-reading program reads the commands for a page, it keeps track of several quantities: (a) The current font f is an integer; this value is changed only by *fnt* and *fnt_num* commands. (b) The current position on the page is given by two numbers called the horizontal and vertical coordinates, h and v. Both coordinates are zero at the upper left corner of the page; moving to the right corresponds to increasing the horizontal coordinate, and moving down corresponds to increasing the vertical coordinate. Thus, the coordinates are essentially Cartesian, except that vertical directions are flipped; the Cartesian version of (h, v) would be $(h, -v)$. (c) The current spacing amounts are given by four numbers w, x, y, and z, where w and x are used for horizontal spacing and where y and z are used for vertical spacing. (d) There is a stack containing (h, v, w, x, y, z) values; the DVI commands *push* and *pop* are used to change the current level of operation. Note that the current font f is not pushed and popped; the stack contains only information about positioning.

The values of h, v, w, x, y, and z are signed integers having up to 32 bits, including the sign. Since they represent physical distances, there is a small unit of measurement such that increasing h by 1 means moving a certain tiny distance to the right. The actual unit of measurement is variable, as explained below.

21. Here is a list of all the commands that may appear in a `DVI` file. Each command is specified by its symbolic name (e.g., *bop*), its opcode byte (e.g., 139), and its parameters (if any). The parameters are followed by a bracketed number telling how many bytes they occupy; for example, '$p[4]$' means that parameter p is four bytes long.

set_char_0 0. Typeset character number 0 from font f such that the reference point of the character is at (h, v). Then increase h by the width of that character. Note that a character may have zero or negative width, so one cannot be sure that h will advance after this command; but h usually does increase.

set_char_1 through *set_char_127* (opcodes 1 to 127). Do the operations of *set_char_0*; but use the character whose number matches the opcode, instead of character 0.

set1 128 $c[1]$. Same as *set_char_0*, except that character number c is typeset. TEX82 uses this command for characters in the range $128 \le c < 256$.

set2 129 $c[2]$. Same as *set1*, except that c is two bytes long, so it is in the range $0 \le c < 65536$.

set3 130 $c[3]$. Same as *set1*, except that c is three bytes long, so it can be as large as $2^{24} - 1$. Not even the Chinese language has this many characters, but this command might prove useful in some yet unforeseen way.

set4 131 $c[4]$. Same as *set1*, except that c is four bytes long, possibly even negative. Imagine that.

set_rule 132 $a[4]$ $b[4]$. Typeset a solid black rectangle of height a and width b, with its bottom left corner at (h, v). Then set $h \leftarrow h + b$. If either $a \le 0$ or $b \le 0$, nothing should be typeset. Note that if $b < 0$, the value of h will decrease even though nothing else happens.

put1 133 $c[1]$. Typeset character number c from font f such that the reference point of the character is at (h, v). (The 'put' commands are exactly like the 'set' commands, except that they simply put out a character or a rule without moving the reference point afterwards.)

put2 134 $c[2]$. Same as *set2*, except that h is not changed.

put3 135 $c[3]$. Same as *set3*, except that h is not changed.

put4 136 $c[4]$. Same as *set4*, except that h is not changed.

put_rule 137 $a[4]$ $b[4]$. Same as *set_rule*, except that h is not changed.

nop 138. No operation, do nothing. Any number of *nop*'s may occur between `DVI` commands, but a *nop* cannot be inserted between a command and its parameters or between two parameters.

bop 139 $c_0[4]$ $c_1[4]$ ... $c_9[4]$ $p[4]$. Beginning of a page: Set $(h, v, w, x, y, z) \leftarrow (0, 0, 0, 0, 0, 0)$ and set the stack empty. Set the current font f to an undefined value. The ten c_i parameters can be used to identify pages, if a user wants to print only part of a `DVI` file; TEX82 gives them the values of `\count0` ... `\count9` at the time `\shipout` was invoked for this page. The parameter p points to the previous *bop* command in the file, where the first *bop* has $p = -1$.

eop 140. End of page: Print what you have read since the previous *bop*. At this point the stack should be empty. (The `DVI`-reading programs that drive most output devices will have kept a buffer of the material that appears on the page that has just ended. This material is largely, but not entirely, in order by v coordinate and (for fixed v) by h coordinate; so it usually needs to be sorted into some order that is appropriate for the device in question. `GFtoDVI` does not do such sorting.)

push 141. Push the current values of (h, v, w, x, y, z) onto the top of the stack; do not change any of these values. Note that f is not pushed.

pop 142. Pop the top six values off of the stack and assign them to (h, v, w, x, y, z). The number of pops should never exceed the number of pushes, since it would be highly embarrassing if the stack were empty at the time of a *pop* command.

right1 143 $b[1]$. Set $h \leftarrow h + b$, i.e., move right b units. The parameter is a signed number in two's complement notation, $-128 \le b < 128$; if $b < 0$, the reference point actually moves left.

right2 144 $b[2]$. Same as *right1*, except that b is a two-byte quantity in the range $-32768 \le b < 32768$.

right3 145 $b[3]$. Same as *right1*, except that b is a three-byte quantity in the range $-2^{23} \le b < 2^{23}$.

right4 146 b[4]. Same as *right1*, except that b is a four-byte quantity in the range $-2^{31} \le b < 2^{31}$.

w0 147. Set $h \leftarrow h + w$; i.e., move right w units. With luck, this parameterless command will usually suffice, because the same kind of motion will occur several times in succession; the following commands explain how w gets particular values.

w1 148 b[1]. Set $w \leftarrow b$ and $h \leftarrow h + b$. The value of b is a signed quantity in two's complement notation, $-128 \le b < 128$. This command changes the current w spacing and moves right by b.

w2 149 b[2]. Same as *w1*, but b is a two-byte-long parameter, $-32768 \le b < 32768$.

w3 150 b[3]. Same as *w1*, but b is a three-byte-long parameter, $-2^{23} \le b < 2^{23}$.

w4 151 b[4]. Same as *w1*, but b is a four-byte-long parameter, $-2^{31} \le b < 2^{31}$.

x0 152. Set $h \leftarrow h + x$; i.e., move right x units. The 'x' commands are like the 'w' commands except that they involve x instead of w.

x1 153 b[1]. Set $x \leftarrow b$ and $h \leftarrow h + b$. The value of b is a signed quantity in two's complement notation, $-128 \le b < 128$. This command changes the current x spacing and moves right by b.

x2 154 b[2]. Same as *x1*, but b is a two-byte-long parameter, $-32768 \le b < 32768$.

x3 155 b[3]. Same as *x1*, but b is a three-byte-long parameter, $-2^{23} \le b < 2^{23}$.

x4 156 b[4]. Same as *x1*, but b is a four-byte-long parameter, $-2^{31} \le b < 2^{31}$.

down1 157 a[1]. Set $v \leftarrow v + a$, i.e., move down a units. The parameter is a signed number in two's complement notation, $-128 \le a < 128$; if $a < 0$, the reference point actually moves up.

down2 158 a[2]. Same as *down1*, except that a is a two-byte quantity in the range $-32768 \le a < 32768$.

down3 159 a[3]. Same as *down1*, except that a is a three-byte quantity in the range $-2^{23} \le a < 2^{23}$.

down4 160 a[4]. Same as *down1*, except that a is a four-byte quantity in the range $-2^{31} \le a < 2^{31}$.

y0 161. Set $v \leftarrow v + y$; i.e., move down y units. With luck, this parameterless command will usually suffice, because the same kind of motion will occur several times in succession; the following commands explain how y gets particular values.

y1 162 a[1]. Set $y \leftarrow a$ and $v \leftarrow v + a$. The value of a is a signed quantity in two's complement notation, $-128 \le a < 128$. This command changes the current y spacing and moves down by a.

y2 163 a[2]. Same as *y1*, but a is a two-byte-long parameter, $-32768 \le a < 32768$.

y3 164 a[3]. Same as *y1*, but a is a three-byte-long parameter, $-2^{23} \le a < 2^{23}$.

y4 165 a[4]. Same as *y1*, but a is a four-byte-long parameter, $-2^{31} \le a < 2^{31}$.

z0 166. Set $v \leftarrow v + z$; i.e., move down z units. The 'z' commands are like the 'y' commands except that they involve z instead of y.

z1 167 a[1]. Set $z \leftarrow a$ and $v \leftarrow v + a$. The value of a is a signed quantity in two's complement notation, $-128 \le a < 128$. This command changes the current z spacing and moves down by a.

z2 168 a[2]. Same as *z1*, but a is a two-byte-long parameter, $-32768 \le a < 32768$.

z3 169 a[3]. Same as *z1*, but a is a three-byte-long parameter, $-2^{23} \le a < 2^{23}$.

z4 170 a[4]. Same as *z1*, but a is a four-byte-long parameter, $-2^{31} \le a < 2^{31}$.

fnt_num_0 171. Set $f \leftarrow 0$. Font 0 must previously have been defined by a *fnt_def* instruction, as explained below.

fnt_num_1 through *fnt_num_63* (opcodes 172 to 234). Set $f \leftarrow 1, \ldots, f \leftarrow 63$, respectively.

fnt1 235 k[1]. Set $f \leftarrow k$. TEX82 uses this command for font numbers in the range $64 \le k < 256$.

fnt2 236 k[2]. Same as *fnt1*, except that k is two bytes long, so it is in the range $0 \le k < 65536$. TEX82 never generates this command, but large font numbers may prove useful for specifications of color or texture, or they may be used for special fonts that have fixed numbers in some external coding scheme.

fnt3 237 k[3]. Same as *fnt1*, except that k is three bytes long, so it can be as large as $2^{24} - 1$.

fnt4 238 $k[4]$. Same as *fnt1*, except that k is four bytes long; this is for the really big font numbers (and for the negative ones).

xxx1 239 $k[1]$ $x[k]$. This command is undefined in general; it functions as a $(k+2)$-byte *nop* unless special `DVI`-reading programs are being used. TEX82 generates *xxx1* when a short enough `\special` appears, setting k to the number of bytes being sent. It is recommended that x be a string having the form of a keyword followed by possible parameters relevant to that keyword.

xxx2 240 $k[2]$ $x[k]$. Like *xxx1*, but $0 \le k < 65536$.

xxx3 241 $k[3]$ $x[k]$. Like *xxx1*, but $0 \le k < 2^{24}$.

xxx4 242 $k[4]$ $x[k]$. Like *xxx1*, but k can be ridiculously large. TEX82 uses *xxx4* when *xxx1* would be incorrect.

fnt_def1 243 $k[1]$ $c[4]$ $s[4]$ $d[4]$ $a[1]$ $l[1]$ $n[a+l]$. Define font k, where $0 \le k < 256$; font definitions will be explained shortly.

fnt_def2 244 $k[2]$ $c[4]$ $s[4]$ $d[4]$ $a[1]$ $l[1]$ $n[a+l]$. Define font k, where $0 \le k < 65536$.

fnt_def3 245 $k[3]$ $c[4]$ $s[4]$ $d[4]$ $a[1]$ $l[1]$ $n[a+l]$. Define font k, where $0 \le k < 2^{24}$.

fnt_def4 246 $k[4]$ $c[4]$ $s[4]$ $d[4]$ $a[1]$ $l[1]$ $n[a+l]$. Define font k, where $-2^{31} \le k < 2^{31}$.

pre 247 $i[1]$ *num*$[4]$ *den*$[4]$ *mag*$[4]$ $k[1]$ $x[k]$. Beginning of the preamble; this must come at the very beginning of the file. Parameters i, *num*, *den*, *mag*, k, and x are explained below.

post 248. Beginning of the postamble, see below.

post_post 249. Ending of the postamble, see below.

Commands 250–255 are undefined at the present time.

22. Only a few of the operation codes above are actually needed by `GFtoDVI`.

```
define set1 = 128  { typeset a character and move right }
define put_rule = 137  { typeset a rule }
define bop = 139  { beginning of page }
define eop = 140  { ending of page }
define push = 141  { save the current positions }
define pop = 142  { restore previous positions }
define right4 = 146  { move right }
define down4 = 160  { move down }
define z0 = 166  { move down z }
define z4 = 170  { move down and set z }
define fnt_num_0 = 171  { set current font to 0 }
define fnt_def1 = 243  { define the meaning of a font number }
define pre = 247  { preamble }
define post = 248  { postamble beginning }
define post_post = 249  { postamble ending }
```

23. The preamble contains basic information about the file as a whole. As stated above, there are six parameters:

$$i[1]\ num[4]\ den[4]\ mag[4]\ k[1]\ x[k].$$

The i byte identifies DVI format; currently this byte is always set to 2. (The value $i = 3$ is currently used for an extended format that allows a mixture of right-to-left and left-to-right typesetting. Some day we will set $i = 4$, when DVI format makes another incompatible change—perhaps in the year 2048.)

The next two parameters, *num* and *den*, are positive integers that define the units of measurement; they are the numerator and denominator of a fraction by which all dimensions in the DVI file could be multiplied in order to get lengths in units of 10^{-7} meters. (For example, there are exactly 7227 TEX points in 254 centimeters, and TEX82 works with scaled points where there are 2^{16} sp in a point, so TEX82 sets $num = 25400000$ and $den = 7227 \cdot 2^{16} = 473628672$.)

The *mag* parameter is what TEX82 calls \mag, i.e., 1000 times the desired magnification. The actual fraction by which dimensions are multiplied is therefore $mag \cdot num/1000den$. Note that if a TEX source document does not call for any '`true`' dimensions, and if you change it only by specifying a different \mag setting, the DVI file that TEX creates will be completely unchanged except for the value of *mag* in the preamble and postamble. (Fancy DVI-reading programs allow users to override the *mag* setting when a DVI file is being printed.)

Finally, k and x allow the DVI writer to include a comment, which is not interpreted further. The length of comment x is k, where $0 \le k < 256$.

define *dvi_id_byte* $= 2$ { identifies the kind of DVI files described here }

24. Font definitions for a given font number k contain further parameters

$$c[4]\ s[4]\ d[4]\ a[1]\ l[1]\ n[a+l].$$

The four-byte value c is the check sum that TEX (or whatever program generated the DVI file) found in the TFM file for this font; c should match the check sum of the font found by programs that read this DVI file.

Parameter s contains a fixed-point scale factor that is applied to the character widths in font k; font dimensions in TFM files and other font files are relative to this quantity, which is always positive and less than 2^{27}. It is given in the same units as the other dimensions of the DVI file. Parameter d is similar to s; it is the "design size," and (like s) it is given in DVI units. Thus, font k is to be used at $mag \cdot s/1000d$ times its normal size.

The remaining part of a font definition gives the external name of the font, which is an ASCII string of length $a + l$. The number a is the length of the "area" or directory, and l is the length of the font name itself; the standard local system font area is supposed to be used when $a = 0$. The n field contains the area in its first a bytes.

Font definitions must appear before the first use of a particular font number. Once font k is defined, it must not be defined again; however, we shall see below that font definitions appear in the postamble as well as in the pages, so in this sense each font number is defined exactly twice, if at all. Like *nop* commands, font definitions can appear before the first *bop*, or between an *eop* and a *bop*.

25. The last page in a DVI file is followed by '*post*'; this command introduces the postamble, which summarizes important facts that TEX has accumulated about the file, making it possible to print subsets of the data with reasonable efficiency. The postamble has the form

post $p[4]$ *num*$[4]$ *den*$[4]$ *mag*$[4]$ $l[4]$ $u[4]$ $s[2]$ $t[2]$
⟨ font definitions ⟩
post_post $q[4]$ $i[1]$ 223's$[\geq 4]$

Here p is a pointer to the final *bop* in the file. The next three parameters, *num*, *den*, and *mag*, are duplicates of the quantities that appeared in the preamble.

Parameters l and u give respectively the height-plus-depth of the tallest page and the width of the widest page, in the same units as other dimensions of the file. These numbers might be used by a DVI-reading program to position individual "pages" on large sheets of film or paper; however, the standard convention for output on normal size paper is to position each page so that the upper left-hand corner is exactly one inch from the left and the top. Experience has shown that it is unwise to design DVI-to-printer software that attempts cleverly to center the output; a fixed position of the upper left corner is easiest for users to understand and to work with. Therefore l and u are often ignored.

Parameter s is the maximum stack depth (i.e., the largest excess of *push* commands over *pop* commands) needed to process this file. Then comes t, the total number of pages (*bop* commands) present.

The postamble continues with font definitions, which are any number of *fnt_def* commands as described above, possibly interspersed with *nop* commands. Each font number that is used in the DVI file must be defined exactly twice: Once before it is first selected by a *fnt* command, and once in the postamble.

26. The last part of the postamble, following the *post_post* byte that signifies the end of the font definitions, contains q, a pointer to the *post* command that started the postamble. An identification byte, i, comes next; this currently equals 2, as in the preamble.

The i byte is followed by four or more bytes that are all equal to the decimal number 223 (i.e., *'337* in octal). TEX puts out four to seven of these trailing bytes, until the total length of the file is a multiple of four bytes, since this works out best on machines that pack four bytes per word; but any number of 223's is allowed, as long as there are at least four of them. In effect, 223 is a sort of signature that is added at the very end.

This curious way to finish off a DVI file makes it feasible for DVI-reading programs to find the postamble first, on most computers, even though TEX wants to write the postamble last. Most operating systems permit random access to individual words or bytes of a file, so the DVI reader can start at the end and skip backwards over the 223's until finding the identification byte. Then it can back up four bytes, read q, and move to byte q of the file. This byte should, of course, contain the value 248 (*post*); now the postamble can be read, so the DVI reader can discover all the information needed for typesetting the pages. Note that it is also possible to skip through the DVI file at reasonably high speed to locate a particular page, if that proves desirable. This saves a lot of time, since DVI files used in production jobs tend to be large.

Unfortunately, however, standard Pascal does not include the ability to access a random position in a file, or even to determine the length of a file. Almost all systems nowadays provide the necessary capabilities, so DVI format has been designed to work most efficiently with modern operating systems.

27. Generic font file format. The "generic font" (GF) input files that GFtoDVI must deal with have a structure that was inspired by DVI format, although the operation codes are quite different in most cases. The term *generic* indicates that this file format doesn't match the conventions of any name-brand manufacturer; but it is easy to convert GF files to the special format required by almost all digital phototypesetting equipment. There's a strong analogy between the DVI files written by TEX and the GF files written by METAFONT; and, in fact, the reader will notice that many of the paragraphs below are identical to their counterparts in the description of DVI already given. The following description has been lifted almost verbatim from the program for METAFONT.

A GF file is a stream of 8-bit bytes that may be regarded as a series of commands in a machine-like language. The first byte of each command is the operation code, and this code is followed by zero or more bytes that provide parameters to the command. The parameters themselves may consist of several consecutive bytes; for example, the '*boc*' (beginning of character) command has six parameters, each of which is four bytes long. Parameters are usually regarded as nonnegative integers; but four-byte-long parameters can be either positive or negative, hence they range in value from -2^{31} to $2^{31}-1$. As in DVI files, numbers that occupy more than one byte position appear in BigEndian order, and negative numbers appear in two's complement notation.

A GF file consists of a "preamble," followed by a sequence of one or more "characters," followed by a "postamble." The preamble is simply a *pre* command, with its parameters that introduce the file; this must come first. Each "character" consists of a *boc* command, followed by any number of other commands that specify "black" pixels, followed by an *eoc* command. The characters appear in the order that METAFONT generated them. If we ignore no-op commands (which are allowed between any two commands in the file), each *eoc* command is immediately followed by a *boc* command, or by a *post* command; in the latter case, there are no more characters in the file, and the remaining bytes form the postamble. Further details about the postamble will be explained later.

Some parameters in GF commands are "pointers." These are four-byte quantities that give the location number of some other byte in the file; the first file byte is number 0, then comes number 1, and so on.

28. The GF format is intended to be both compact and easily interpreted by a machine. Compactness is achieved by making most of the information relative instead of absolute. When a GF-reading program reads the commands for a character, it keeps track of two quantities: (a) the current column number, m; and (b) the current row number, n. These are 32-bit signed integers, although most actual font formats produced from GF files will need to curtail this vast range because of practical limitations. (METAFONT output will never allow $|m|$ or $|n|$ to get extremely large, but the GF format tries to be more general.)

How do GF's row and column numbers correspond to the conventions of TEX and METAFONT? Well, the "reference point" of a character, in TEX's view, is considered to be at the lower left corner of the pixel in row 0 and column 0. This point is the intersection of the baseline with the left edge of the type; it corresponds to location $(0,0)$ in METAFONT programs. Thus the pixel in GF row 0 and column 0 is METAFONT's unit square, comprising the region of the plane whose coordinates both lie between 0 and 1. The pixel in GF row n and column m consists of the points whose METAFONT coordinates (x,y) satisfy $m \le x \le m+1$ and $n \le y \le n+1$. Negative values of m and x correspond to columns of pixels *left* of the reference point; negative values of n and y correspond to rows of pixels *below* the baseline.

Besides m and n, there's also a third aspect of the current state, namely the *paint_switch*, which is always either *black* or *white*. Each *paint* command advances m by a specified amount d, and blackens the intervening pixels if *paint_switch* = *black*; then the *paint_switch* changes to the opposite state. GF's commands are designed so that m will never decrease within a row, and n will never increase within a character; hence there is no way to whiten a pixel that has been blackened.

29. Here is a list of all the commands that may appear in a GF file. Each command is specified by its symbolic name (e.g., *boc*), its opcode byte (e.g., 67), and its parameters (if any). The parameters are followed by a bracketed number telling how many bytes they occupy; for example, '$d[2]$' means that parameter d is two bytes long.

paint_0 0. This is a *paint* command with $d = 0$; it does nothing but change the *paint_switch* from *black* to *white* or vice versa.

paint_1 through *paint_63* (opcodes 1 to 63). These are *paint* commands with $d = 1$ to 63, defined as follows: If *paint_switch* = *black*, blacken d pixels of the current row n, in columns m through $m + d - 1$ inclusive. Then, in any case, complement the *paint_switch* and advance m by d.

paint1 64 $d[1]$. This is a *paint* command with a specified value of d; METAFONT uses it to paint when $64 \le d < 256$.

paint2 65 $d[2]$. Same as *paint1*, but d can be as high as 65535.

paint3 66 $d[3]$. Same as *paint1*, but d can be as high as $2^{24} - 1$. METAFONT never needs this command, and it is hard to imagine anybody making practical use of it; surely a more compact encoding will be desirable when characters can be this large. But the command is there, anyway, just in case.

boc 67 $c[4]$ $p[4]$ *min_m*$[4]$ *max_m*$[4]$ *min_n*$[4]$ *max_n*$[4]$. Beginning of a character: Here c is the character code, and p points to the previous character beginning (if any) for characters having this code number modulo 256. (The pointer p is -1 if there was no prior character with an equivalent code.) The values of registers m and n defined by the instructions that follow for this character must satisfy *min_m* $\le m \le$ *max_m* and *min_n* $\le n \le$ *max_n*. (The values of *max_m* and *min_n* need not be the tightest bounds possible.) When a GF-reading program sees a *boc*, it can use *min_m*, *max_m*, *min_n*, and *max_n* to initialize the bounds of an array. Then it sets $m \leftarrow$ *min_m*, $n \leftarrow$ *max_n*, and *paint_switch* $\leftarrow$ *white*.

boc1 68 $c[1]$ *del_m*$[1]$ *max_m*$[1]$ *del_n*$[1]$ *max_n*$[1]$. Same as *boc*, but p is assumed to be -1; also *del_m* = *max_m* − *min_m* and *del_n* = *max_n* − *min_n* are given instead of *min_m* and *min_n*. The one-byte parameters must be between 0 and 255, inclusive. (This abbreviated *boc* saves 19 bytes per character, in common cases.)

eoc 69. End of character: All pixels blackened so far constitute the pattern for this character. In particular, a completely blank character might have *eoc* immediately following *boc*.

skip0 70. Decrease n by 1 and set $m \leftarrow$ *min_m*, *paint_switch* $\leftarrow$ *white*. (This finishes one row and begins another, ready to whiten the leftmost pixel in the new row.)

skip1 71 $d[1]$. Decrease n by $d + 1$, set $m \leftarrow$ *min_m*, and set *paint_switch* $\leftarrow$ *white*. This is a way to produce d all-white rows.

skip2 72 $d[2]$. Same as *skip1*, but d can be as large as 65535.

skip3 73 $d[3]$. Same as *skip1*, but d can be as large as $2^{24} - 1$. METAFONT obviously never needs this command.

new_row_0 74. Decrease n by 1 and set $m \leftarrow$ *min_m*, *paint_switch* $\leftarrow$ *black*. (This finishes one row and begins another, ready to *blacken* the leftmost pixel in the new row.)

new_row_1 through *new_row_164* (opcodes 75 to 238). Same as *new_row_0*, but with $m \leftarrow$ *min_m* + 1 through *min_m* + 164, respectively.

xxx1 239 $k[1]$ $x[k]$. This command is undefined in general; it functions as a $(k + 2)$-byte *no_op* unless special GF-reading programs are being used. METAFONT generates *xxx* commands when encountering a **special** string; this occurs in the GF file only between characters, after the preamble, and before the postamble. However, *xxx* commands might appear within characters, in GF files generated by other processors. It is recommended that x be a string having the form of a keyword followed by possible parameters relevant to that keyword.

xxx2 240 $k[2]$ $x[k]$. Like *xxx1*, but $0 \le k < 65536$.

xxx3 241 $k[3]$ $x[k]$. Like *xxx1*, but $0 \le k < 2^{24}$. METAFONT uses this when sending a **special** string whose length exceeds 255.

xxx4 242 $k[4]$ $x[k]$. Like *xxx1*, but k can be ridiculously large; k mustn't be negative.

yyy 243 $y[4]$. This command is undefined in general; it functions as a 5-byte *no_op* unless special GF-reading programs are being used. METAFONT puts *scaled* numbers into *yyy*'s, as a result of **numspecial** commands; the intent is to provide numeric parameters to *xxx* commands that immediately precede.

no_op 244. No operation, do nothing. Any number of *no_op*'s may occur between GF commands, but a *no_op* cannot be inserted between a command and its parameters or between two parameters.

char_loc 245 $c[1]$ $dx[4]$ $dy[4]$ $w[4]$ $p[4]$. This command will appear only in the postamble, which will be explained shortly.

char_loc0 246 $c[1]$ $dm[1]$ $w[4]$ $p[4]$. Same as *char_loc*, except that dy is assumed to be zero, and the value of dx is taken to be $65536 * dm$, where $0 \le dm < 256$.

pre 247 $i[1]$ $k[1]$ $x[k]$. Beginning of the preamble; this must come at the very beginning of the file. Parameter i is an identifying number for GF format, currently 131. The other information is merely commentary; it is not given special interpretation like *xxx* commands are. (Note that *xxx* commands may immediately follow the preamble, before the first *boc*.)

post 248. Beginning of the postamble, see below.

post_post 249. Ending of the postamble, see below.

Commands 250–255 are undefined at the present time.

define *gf_id_byte* = 131 { identifies the kind of GF files described here }

30. Here are the opcodes that GFtoDVI actually refers to.

```
define paint_0 = 0 { beginning of the paint commands }
define paint1 = 64 { move right a given number of columns, then black ↔ white }
define paint2 = 65 { ditto, with potentially larger number of columns }
define paint3 = 66 { ditto, with potentially excessive number of columns }
define boc = 67 { beginning of a character }
define boc1 = 68 { abbreviated boc }
define eoc = 69 { end of a character }
define skip0 = 70 { skip no blank rows }
define skip1 = 71 { skip over blank rows }
define skip2 = 72 { skip over lots of blank rows }
define skip3 = 73 { skip over a huge number of blank rows }
define new_row_0 = 74 { move down one row and then right }
define xxx1 = 239 { for special strings }
define xxx2 = 240 { for somewhat long special strings }
define xxx3 = 241 { for extremely long special strings }
define xxx4 = 242 { for incredibly long special strings }
define yyy = 243 { for numspecial numbers }
define no_op = 244 { no operation }
```

31. The last character in a GF file is followed by '*post*'; this command introduces the postamble, which summarizes important facts that METAFONT has accumulated. The postamble has the form

post *p*[4] *ds*[4] *cs*[4] *hppp*[4] *vppp*[4] *min_m*[4] *max_m*[4] *min_n*[4] *max_n*[4]
⟨ character locators ⟩
post_post *q*[4] *i*[1] 223's[≥4]

Here p is a pointer to the byte following the final *eoc* in the file (or to the byte following the preamble, if there are no characters); it can be used to locate the beginning of *xxx* commands that might have preceded the postamble. The *ds* and *cs* parameters give the design size and check sum, respectively, of the font (see the description of TFM format below). Parameters *hppp* and *vppp* are the ratios of pixels per point, horizontally and vertically, expressed as *scaled* integers (i.e., multiplied by 2^{16}); they can be used to correlate the font with specific device resolutions, magnifications, and "at sizes." Then come *min_m*, *max_m*, *min_n*, and *max_n*, which bound the values that registers m and n assume in all characters in this GF file. (These bounds need not be the best possible; *max_m* and *min_n* may, on the other hand, be tighter than the similar bounds in *boc* commands. For example, some character may have $min\_n = -100$ in its *boc*, but it might turn out that n never gets lower than -50 in any character; then *min_n* can have any value ≤ -50. If there are no characters in the file, it's possible to have $min\_m > max\_m$ and/or $min\_n > max\_n$.)

32. Character locators are introduced by *char_loc* commands, which specify a character residue c, character escapements (dx, dy), a character width w, and a pointer p to the beginning of that character. (If two or more characters have the same code c modulo 256, only the last will be indicated; the others can be located by following backpointers. Characters whose codes differ by a multiple of 256 are assumed to share the same font metric information, hence the TFM file contains only residues of character codes modulo 256. This convention is intended for oriental languages, when there are many character shapes but few distinct widths.)

The character escapements (dx, dy) are the values of METAFONT's **chardx** and **chardy** parameters; they are in units of *scaled* pixels; i.e., dx is in horizontal pixel units times 2^{16}, and dy is in vertical pixel units times 2^{16}. This is the intended amount of displacement after typesetting the character; for DVI files, dy should be zero, but other document file formats allow nonzero vertical escapement.

The character width w duplicates the information in the TFM file; it is 2^{20} times the ratio of the true width to the font's design size.

The backpointer p points to the character's *boc*, or to the first of a sequence of consecutive *xxx* or *yyy* or *no_op* commands that immediately precede the *boc*, if such commands exist; such "special" commands essentially belong to the characters, while the special commands after the final character belong to the postamble (i.e., to the font as a whole). This convention about p applies also to the backpointers in *boc* commands, even though it wasn't explained in the description of *boc*.

Pointer p might be -1 if the character exists in the TFM file but not in the GF file. This unusual situation can arise in METAFONT output if the user had $proofing < 0$ when the character was being shipped out, but then made $proofing \geq 0$ in order to get a GF file.

33. The last part of the postamble, following the *post_post* byte that signifies the end of the character locators, contains q, a pointer to the *post* command that started the postamble. An identification byte, i, comes next; this currently equals 131, as in the preamble.

The i byte is followed by four or more bytes that are all equal to the decimal number 223 (i.e., *'337* in octal). METAFONT puts out four to seven of these trailing bytes, until the total length of the file is a multiple of four bytes, since this works out best on machines that pack four bytes per word; but any number of 223's is allowed, as long as there are at least four of them. In effect, 223 is a sort of signature that is added at the very end.

This curious way to finish off a `GF` file makes it feasible for `GF`-reading programs to find the postamble first, on most computers, even though METAFONT wants to write the postamble last. Most operating systems permit random access to individual words or bytes of a file, so the `GF` reader can start at the end and skip backwards over the 223's until finding the identification byte. Then it can back up four bytes, read q, and move to byte q of the file. This byte should, of course, contain the value 248 (*post*); now the postamble can be read, so the `GF` reader can discover all the information needed for individual characters.

Unfortunately, however, standard Pascal does not include the ability to access a random position in a file, or even to determine the length of a file. Almost all systems nowadays provide the necessary capabilities, so `GF` format has been designed to work most efficiently with modern operating systems. But if `GF` files have to be processed under the restrictions of standard Pascal, one can simply read them from front to back. This will be adequate for most applications. However, the postamble-first approach would facilitate a program that merges two `GF` files, replacing data from one that is overridden by corresponding data in the other.

34. Extensions to the generic format. The *xxx* and *yyy* instructions understood by GFtoDVI will be listed now, so that we have a convenient reference to all of the special assumptions made later.

Each special instruction begins with an *xxx* command, which consists of either a keyword by itself, or a keyword followed by a space followed by arguments. This *xxx* command may then be followed by *yyy* commands that are understood to be arguments.

The keywords of special instructions that are intended to be used at many different sites should be published as widely as possible in order to minimize conflicts. The first person to establish a keyword presumably has a right to define it; GFtoDVI, as the first program to use extended GF commands, has the opportunity of choosing any keywords it likes, and the responsibility of choosing reasonable ones. Since labels are expected to account for the bulk of extended commands in typical uses of METAFONT, the "null" keyword has been set aside to denote a labeling command.

35. Here then are the special commands of GFtoDVI.

␣n⟨string⟩ *x y*. Here n denotes the type of label; the characters 1, 2, 3, 4 respectively denote labels forced to be at the top, left, right, or bottom of their dot, and the characters 5, 6, 7, 8 stand for the same possibilities but with no dot printed. The character 0 instructs GFtoDVI to choose one of the first four possibilities, if there's no overlap with other labels or dots, otherwise an "overflow" entry is placed at the right of the figure. The character / is the same as 0 except that overflow entries are not produced. The label itself is the ⟨string⟩ that follows. METAFONT coordinates of the point that is to receive this label are given by arguments x and y, in units of scaled pixels. (These arguments appear in *yyy* commands.) (Precise definitions of the size and positioning of labels, and of the notion of "conflicting" labels, will be given later.)

rule x_1 y_1 x_2 y_2. This command draws a line from (x_1, y_1) to (x_2, y_2) in METAFONT coordinates. The present implementation does this only if the line is either horizontal or vertical, or if its slope matches the slope of the slant font.

title␣⟨string⟩. This command (which is output by METAFONT when it sees a "title statement") specifies a string that will appear at the top of the next proofsheet to be output by GFtoDVI. If more than one title is given, they will appear in sequence; titles should be short enough to fit on a single line.

titlefont␣⟨string⟩. This command, and the other font-naming commands below, must precede the first *boc* in the GF file. It overrides the current font used to typeset the titles at the top of proofsheets. GFtoDVI has default fonts that will be used if none other are specified; the "current" title font is initially the default title font.

titlefontarea␣⟨string⟩. This command overrides the current file area (or directory name) from which GFtoDVI will try to find metric information for the title font.

titlefontat *s*. This command overrides the current "at size" that will be used for the title font. (See the discussion of font metric files below, for the meaning of "at size" versus "design size.") The value of s is given in units of scaled points.

labelfont␣⟨string⟩. This command overrides the current font used to typeset the labels that are superimposed on proof figures. (The label font is fairly arbitrary, but it should be dark enough to stand out when superimposed on gray pixels, and it should contain at least the decimal digits and the characters '(', ')', '=', '+', '−', ',', and '.'.)

labelfontarea␣⟨string⟩. This command overrides the current file area (or directory name) from which GFtoDVI will try to find metric information for the label font.

labelfontat *s*. This command overrides the current "at size" that will be used for the label font.

grayfont␣⟨string⟩. This command overrides the current font used to typeset the black pixels and the dots for labels. (Gray fonts will be explained in detail later.)

grayfontarea␣⟨string⟩. This command overrides the current file area (or directory name) from which GFtoDVI will try to find metric information for the gray font.

grayfontat *s*. This command overrides the current "at size" that will be used for the gray font.

`slantfont␣`⟨ string ⟩. This command overrides the current font used to typeset rules that are neither horizontal nor vertical. (Slant fonts will be explained in detail later.)

`slantfontarea␣`⟨ string ⟩. This command overrides the current file area (or directory name) from which `GFtoDVI` will try to find metric information for the slant font.

`slantfontat` s. This command overrides the current "at size" that will be used for the slant font.

`rulethickness` t. This command overrides the current value used for the thickness of rules. If the current value is negative, no rule will be drawn; if the current value is zero, the rule thickness will be specified by a parameter of the gray font. Each `rule` command uses the rule thickness that is current at the time the command appears; hence it is possible to get different thicknesses of rules on the same figure. The value of t is given in units of scaled points (TEX's '`sp`'). At the beginning of each character the current rule thickness is zero.

`offset` x y. This command overrides the current offset values that are added to all coordinates of a character being output; x and y are given as scaled METAFONT coordinates. This simply has the effect of repositioning the figures on the pages; the title line always appears in the same place, but the figure can be moved up, down, left, or right. At the beginning of each character the current offsets are zero.

`xoffset` x. This command is output by METAFONT just before shipping out a character whose x offset is nonzero. `GFtoDVI` adds the specified amount to the x coordinates of all dots, labels, and rules in the following character.

`yoffset` y. This command is output by METAFONT just before shipping out a character whose y offset is nonzero. `GFtoDVI` adds the specified amount to the y coordinates of all dots, labels, and rules in the following character.

36. Font metric data. Before we can get into the meaty details of GFtoDVI, we need to deal with yet another esoteric binary file format, since GFtoDVI also does elementary typesetting operations. Therefore it has to read important information about the fonts it will be using. The following material (again copied almost verbatim from TEX) describes the contents of so-called TEX font metric (TFM) files.

The idea behind TFM files is that typesetting routines need a compact way to store the relevant information about fonts, and computer centers need a compact way to store the relevant information about several hundred fonts. TFM files are compact, and most of the information they contain is highly relevant, so they provide a solution to the problem. GFtoDVI uses only four fonts, but interesting changes in its output will occur when those fonts are varied.

The information in a TFM file appears in a sequence of 8-bit bytes. Since the number of bytes is always a multiple of 4, we could also regard the file as a sequence of 32-bit words; but TEX uses the byte interpretation, and so does GFtoDVI. The individual bytes are considered to be unsigned numbers.

37. The first 24 bytes (6 words) of a TFM file contain twelve 16-bit integers that give the lengths of the various subsequent portions of the file. These twelve integers are, in order:

lf = length of the entire file, in words;
lh = length of the header data, in words;
bc = smallest character code in the font;
ec = largest character code in the font;
nw = number of words in the width table;
nh = number of words in the height table;
nd = number of words in the depth table;
ni = number of words in the italic correction table;
nl = number of words in the lig/kern table;
nk = number of words in the kern table;
ne = number of words in the extensible character table;
np = number of font parameter words.

They are all nonnegative and less than 2^{15}. We must have $bc - 1 \le ec \le 255$, and

$$lf = 6 + lh + (ec - bc + 1) + nw + nh + nd + ni + nl + nk + ne + np.$$

Note that a font may contain as many as 256 characters (if $bc = 0$ and $ec = 255$), and as few as 0 characters (if $bc = ec + 1$). When two or more 8-bit bytes are combined to form an integer of 16 or more bits, the bytes appear in BigEndian order.

⟨ Globals in the outer block 12 ⟩ +≡
$lf, lh, bc, ec, nw, nh, nd, ni, nl, nk, ne, np$: 0 .. ´77777; { subfile sizes }

38. The rest of the TFM file may be regarded as a sequence of ten data arrays having the informal specification

$$\begin{aligned}
\mathit{header} &: \textbf{array}\ [0\,..\,lh-1]\ \textbf{of}\ \mathit{stuff}\\
\mathit{char_info} &: \textbf{array}\ [bc\,..\,ec]\ \textbf{of}\ \mathit{char_info_word}\\
\mathit{width} &: \textbf{array}\ [0\,..\,nw-1]\ \textbf{of}\ \mathit{fix_word}\\
\mathit{height} &: \textbf{array}\ [0\,..\,nh-1]\ \textbf{of}\ \mathit{fix_word}\\
\mathit{depth} &: \textbf{array}\ [0\,..\,nd-1]\ \textbf{of}\ \mathit{fix_word}\\
\mathit{italic} &: \textbf{array}\ [0\,..\,ni-1]\ \textbf{of}\ \mathit{fix_word}\\
\mathit{lig_kern} &: \textbf{array}\ [0\,..\,nl-1]\ \textbf{of}\ \mathit{lig_kern_command}\\
\mathit{kern} &: \textbf{array}\ [0\,..\,nk-1]\ \textbf{of}\ \mathit{fix_word}\\
\mathit{exten} &: \textbf{array}\ [0\,..\,ne-1]\ \textbf{of}\ \mathit{extensible_recipe}\\
\mathit{param} &: \textbf{array}\ [1\,..\,np]\ \textbf{of}\ \mathit{fix_word}
\end{aligned}$$

The most important data type used here is a *fix_word*, which is a 32-bit representation of a binary fraction. A *fix_word* is a signed quantity, with the two's complement of the entire word used to represent negation. Of the 32 bits in a *fix_word*, exactly 12 are to the left of the binary point; thus, the largest *fix_word* value is $2048 - 2^{-20}$, and the smallest is -2048. We will see below, however, that all but two of the *fix_word* values must lie between -16 and $+16$.

39. The first data array is a block of header information, which contains general facts about the font. The header must contain at least two words, and for TFM files to be used with Xerox printing software it must contain at least 18 words, allocated as described below. When different kinds of devices need to be interfaced, it may be necessary to add further words to the header block.

header[0] is a 32-bit check sum that GFtoDVI will copy into the DVI output file whenever it uses the font. Later on when the DVI file is printed, possibly on another computer, the actual font that gets used is supposed to have a check sum that agrees with the one in the TFM file used by GFtoDVI. In this way, users will be warned about potential incompatibilities. (However, if the check sum is zero in either the font file or the TFM file, no check is made.) The actual relation between this check sum and the rest of the TFM file is not important; the check sum is simply an identification number with the property that incompatible fonts almost always have distinct check sums.

header[1] is a *fix_word* containing the design size of the font, in units of TEX points (7227 TEX points = 254 cm). This number must be at least 1.0; it is fairly arbitrary, but usually the design size is 10.0 for a "10 point" font, i.e., a font that was designed to look best at a 10-point size, whatever that really means. When a TEX user asks for a font '`at` δ `pt`', the effect is to override the design size and replace it by δ, and to multiply the x and y coordinates of the points in the font image by a factor of δ divided by the design size. Similarly, specific sizes can be substituted for the design size by GFtoDVI commands like '`titlefontat`'. *All other dimensions in the* TFM *file are fix_word numbers in design-size units.* Thus, for example, the value of *param*[6], one `em` or `\quad`, is often the *fix_word* value $2^{20} = 1.0$, since many fonts have a design size equal to one em. The other dimensions must be less than 16 design-size units in absolute value; thus, *header*[1] and *param*[1] are the only *fix_word* entries in the whole TFM file whose first byte might be something besides 0 or 255.

header[2 .. 11], if present, contains 40 bytes that identify the character coding scheme. The first byte, which must be between 0 and 39, is the number of subsequent ASCII bytes actually relevant in this string, which is intended to specify what character-code-to-symbol convention is present in the font. Examples are `ASCII` for standard ASCII, `TeX text` for fonts like `cmr10` and `cmti9`, `TeX math extension` for `cmex10`, `XEROX text` for Xerox fonts, `GRAPHIC` for special-purpose non-alphabetic fonts, `GFGRAY` for GFtoDVI's gray fonts, `GFSLANT` for GFtoDVI's slant fonts, `UNSPECIFIED` for the default case when there is no information. Parentheses should not appear in this name. (Such a string is said to be in BCPL format.)

header[12 .. whatever] might also be present.

40. Next comes the *char_info* array, which contains one *char_info_word* per character. Each *char_info_word* contains six fields packed into four bytes as follows.

first byte: *width_index* (8 bits)
second byte: *height_index* (4 bits) times 16, plus *depth_index* (4 bits)
third byte: *italic_index* (6 bits) times 4, plus *tag* (2 bits)
fourth byte: *remainder* (8 bits)

The actual width of a character is *width*[*width_index*], in design-size units; this is a device for compressing information, since many characters have the same width. Since it is quite common for many characters to have the same height, depth, or italic correction, the TFM format imposes a limit of 16 different heights, 16 different depths, and 64 different italic corrections.

Incidentally, the relation *width*[0] = *height*[0] = *depth*[0] = *italic*[0] = 0 should always hold, so that an index of zero implies a value of zero. The *width_index* should never be zero unless the character does not exist in the font, since a character is valid if and only if it lies between *bc* and *ec* and has a nonzero *width_index*.

41. The *tag* field in a *char_info_word* has four values that explain how to interpret the *remainder* field.

tag = 0 (*no_tag*) means that *remainder* is unused.
tag = 1 (*lig_tag*) means that this character has a ligature/kerning program starting at *lig_kern*[*remainder*].
tag = 2 (*list_tag*) means that this character is part of a chain of characters of ascending sizes, and not the largest in the chain. The *remainder* field gives the character code of the next larger character.
tag = 3 (*ext_tag*) means that this character code represents an extensible character, i.e., a character that is built up of smaller pieces so that it can be made arbitrarily large. The pieces are specified in *exten*[*remainder*].

define *no_tag* = 0 { vanilla character }
define *lig_tag* = 1 { character has a ligature/kerning program }
define *list_tag* = 2 { character has a successor in a charlist }
define *ext_tag* = 3 { character is extensible }

42. The *lig_kern* array contains instructions in a simple programming language that explains what to do for special letter pairs. Each word in this array is a *lig_kern_command* of four bytes.

first byte: *skip_byte*, indicates that this is the final program step if the byte is 128 or more, otherwise the next step is obtained by skipping this number of intervening steps.

second byte: *next_char*, "if *next_char* follows the current character, then perform the operation and stop, otherwise continue."

third byte: *op_byte*, indicates a ligature step if less than 128, a kern step otherwise.

fourth byte: *remainder*.

In a kern step, an additional space equal to $kern[256 * (op\_byte - 128) + remainder]$ is inserted between the current character and *next_char*. This amount is often negative, so that the characters are brought closer together by kerning; but it might be positive.

There are eight kinds of ligature steps, having *op_byte* codes $4a+2b+c$ where $0 \le a \le b+c$ and $0 \le b, c \le 1$. The character whose code is *remainder* is inserted between the current character and *next_char*; then the current character is deleted if $b = 0$, and *next_char* is deleted if $c = 0$; then we pass over a characters to reach the next current character (which may have a ligature/kerning program of its own).

If the very first instruction of the *lig_kern* array has *skip_byte* = 255, the *next_char* byte is the so-called right boundary character of this font; the value of *next_char* need not lie between *bc* and *ec*. If the very last instruction of the *lig_kern* array has *skip_byte* = 255, there is a special ligature/kerning program for a left boundary character, beginning at location $256 * op\_byte + remainder$. The interpretation is that TEX puts implicit boundary characters before and after each consecutive string of characters from the same font. These implicit characters do not appear in the output, but they can affect ligatures and kerning.

If the very first instruction of a character's *lig_kern* program has *skip_byte* > 128, the program actually begins in location $256 * op\_byte + remainder$. This feature allows access to large *lig_kern* arrays, because the first instruction must otherwise appear in a location ≤ 255.

Any instruction with *skip_byte* > 128 in the *lig_kern* array must have $256 * op\_byte + remainder < nl$. If such an instruction is encountered during normal program execution, it denotes an unconditional halt; no ligature or kerning command is performed.

define *stop_flag* = 128 { value indicating 'STOP' in a lig/kern program }
define *kern_flag* = 128 { op code for a kern step }

43. Extensible characters are specified by an *extensible_recipe*, which consists of four bytes called *top*, *mid*, *bot*, and *rep* (in this order). These bytes are the character codes of individual pieces used to build up a large symbol. If *top*, *mid*, or *bot* are zero, they are not present in the built-up result. For example, an extensible vertical line is like an extensible bracket, except that the top and bottom pieces are missing.

44. The final portion of a TFM file is the *param* array, which is another sequence of *fix_word* values.

$param[1]$ = *slant* is the amount of italic slant. For example, *slant* = .25 means that when you go up one unit, you also go .25 units to the right. The *slant* is a pure number; it's the only *fix_word* other than the design size itself that is not scaled by the design size.

$param[2]$ = *space* is the normal spacing between words in text. Note that character "␣" in the font need not have anything to do with blank spaces.

$param[3]$ = *space_stretch* is the amount of glue stretching between words.

$param[4]$ = *space_shrink* is the amount of glue shrinking between words.

$param[5]$ = *x_height* is the height of letters for which accents don't have to be raised or lowered.

$param[6]$ = *quad* is the size of one em in the font.

$param[7]$ = *extra_space* is the amount added to $param[2]$ at the ends of sentences.

When the character coding scheme is GFGRAY or GFSLANT, the font is supposed to contain an additional parameter called *default_rule_thickness*. Other special parameters go with other coding schemes.

45. Input from binary files. We have seen that GF and DVI and TFM files are sequences of 8-bit bytes. The bytes appear physically in what is called a '**packed file of** 0 .. 255' in Pascal lingo.

Packing is system dependent, and many Pascal systems fail to implement such files in a sensible way (at least, from the viewpoint of producing good production software). For example, some systems treat all byte-oriented files as text, looking for end-of-line marks and such things. Therefore some system-dependent code is often needed to deal with binary files, even though most of the program in this section of GFtoDVI is written in standard Pascal.

One common way to solve the problem is to consider files of *integer* numbers, and to convert an integer in the range $-2^{31} \le x < 2^{31}$ to a sequence of four bytes (a, b, c, d) using the following code, which avoids the controversial integer division of negative numbers:

```
if x ≥ 0 then a ← x div ´100000000
else begin x ← (x + ´10000000000) + ´10000000000; a ← x div ´100000000 + 128;
  end ;
x ← x mod ´100000000;
b ← x div ´200000; x ← x mod ´200000;
c ← x div ´400; d ← x mod ´400;
```

The four bytes are then kept in a buffer and output one by one. (On 36-bit computers, an additional division by 16 is necessary at the beginning. Another way to separate an integer into four bytes is to use/abuse Pascal's variant records, storing an integer and retrieving bytes that are packed in the same place; *caveat implementor!*) It is also desirable in some cases to read a hundred or so integers at a time, maintaining a larger buffer.

We shall stick to simple Pascal in this program, for reasons of clarity, even if such simplicity is sometimes unrealistic.

```
⟨ Types in the outer block 9 ⟩ +≡
  eight_bits = 0 .. 255;  { unsigned one-byte quantity }
  byte_file = packed file of eight_bits;  { files that contain binary data }
```

46. The program deals with three binary file variables: *gf_file* is the main input file that we are converting into a document; *dvi_file* is the main output file that will specify that document; and *tfm_file* is the current font metric file from which character-width information is being read.

```
⟨ Globals in the outer block 12 ⟩ +≡
gf_file: byte_file;  { the character data we are reading }
dvi_file: byte_file;  { the typesetting instructions we are writing }
tfm_file: byte_file;  { a font metric file }
```

47. To prepare these files for input or output, we *reset* or *rewrite* them. An extension of Pascal is needed, since we want to associate it with external files whose names are specified dynamically (i.e., not known at compile time). The following code assumes that '*reset*(f, s)' and '*rewrite*(f, s)' do this, when f is a file variable and s is a string variable that specifies the file name.

```
procedure open_gf_file;  { prepares to read packed bytes in gf_file }
  begin reset(gf_file, name_of_file); cur_loc ← 0;
  end;
procedure open_tfm_file;  { prepares to read packed bytes in tfm_file }
  begin reset(tfm_file, name_of_file);
  end;
procedure open_dvi_file;  { prepares to write packed bytes in dvi_file }
  begin rewrite(dvi_file, name_of_file);
  end;
```

48. If you looked carefully at the preceding code, you probably asked, "What are *cur_loc* and *name_of_file*?" Good question. They are global variables: The integer *cur_loc* tells which byte of the input file will be read next, and the string *name_of_file* will be set to the current file name before the file-opening procedures are called.

```
⟨ Globals in the outer block 12 ⟩ +≡
cur_loc: integer;  { current byte number in gf_file }
name_of_file: packed array [1 .. file_name_size] of char;  { external file name }
```

49. It turns out to be convenient to read four bytes at a time, when we are inputting from TFM files. The input goes into global variables *b0*, *b1*, *b2*, and *b3*, with *b0* getting the first byte and *b3* the fourth.

```
⟨ Globals in the outer block 12 ⟩ +≡
b0, b1, b2, b3: eight_bits;  { four bytes input at once }
```

50. The *read_tfm_word* procedure sets *b0* through *b3* to the next four bytes in the current TFM file.

```
procedure read_tfm_word;
  begin read(tfm_file, b0); read(tfm_file, b1); read(tfm_file, b2); read(tfm_file, b3);
  end;
```

51. We shall use another set of simple functions to read the next byte or bytes from *gf_file*. There are four possibilities, each of which is treated as a separate function in order to minimize the overhead for subroutine calls.

```
function get_byte: integer;  { returns the next byte, unsigned }
  var b: eight_bits;
  begin if eof(gf_file) then get_byte ← 0
  else begin read(gf_file, b); incr(cur_loc); get_byte ← b;
    end;
  end;
function get_two_bytes: integer;  { returns the next two bytes, unsigned }
  var a, b: eight_bits;
  begin read(gf_file, a); read(gf_file, b); cur_loc ← cur_loc + 2; get_two_bytes ← a * 256 + b;
  end;
function get_three_bytes: integer;  { returns the next three bytes, unsigned }
  var a, b, c: eight_bits;
  begin read(gf_file, a); read(gf_file, b); read(gf_file, c); cur_loc ← cur_loc + 3;
  get_three_bytes ← (a * 256 + b) * 256 + c;
  end;
function signed_quad: integer;  { returns the next four bytes, signed }
  var a, b, c, d: eight_bits;
  begin read(gf_file, a); read(gf_file, b); read(gf_file, c); read(gf_file, d); cur_loc ← cur_loc + 4;
  if a < 128 then signed_quad ← ((a * 256 + b) * 256 + c) * 256 + d
  else signed_quad ← (((a − 256) * 256 + b) * 256 + c) * 256 + d;
  end;
```

52. Reading the font information. Now let's get down to brass tacks and consider the more substantial routines that actually convert TFM data into a form suitable for computation. The routines in this part of the program have been borrowed from TEX, with slight changes, since GFtoDVI has to do some of the things that TEX does.

The TFM data is stored in a large array called *font_info*. Each item of *font_info* is a *memory_word*; the *fix_word* data gets converted into *scaled* entries, while everything else goes into words of type *four_quarters*. (These data structures are special cases of the more general memory words of TEX. On some machines it is necessary to define *min_quarterword* = −128 and *max_quarterword* = 127 in order to pack four quarterwords into a single word.)

```
define min_quarterword = 0  { change this to allow efficient packing, if necessary }
define max_quarterword = 255  { ditto }
define qi(#) ≡ # + min_quarterword  { to put an eight_bits item into a quarterword }
define qo(#) ≡ # − min_quarterword  { to take an eight_bits item out of a quarterword }
define title_font = 1
define label_font = 2
define gray_font = 3
define slant_font = 4
define logo_font = 5
define non_char ≡ qi(256)
define non_address ≡ font_mem_size
```

⟨ Types in the outer block 9 ⟩ +≡

```
font_index = 0 .. font_mem_size; quarterword = min_quarterword .. max_quarterword;  { 1/4 of a word }
four_quarters = packed record b0: quarterword;
  b1: quarterword;
  b2: quarterword;
  b3: quarterword;
  end;
memory_word = record
  case boolean of
  true: (sc : scaled);
  false: (qqqq : four_quarters);
  end;
internal_font_number = title_font .. logo_font;
```

53. Besides *font_info*, there are also a number of index arrays that point into it, so that we can locate width and height information, etc. For example, the *char_info* data for character c in font f will be in $font\_info[char\_base[f]+c].qqqq$; and if w is the *width_index* part of this word (the *b0* field), the width of the character is $font\_info[width\_base[f]+w].sc$. (These formulas assume that *min_quarterword* has already been added to w, but not to c.)

```
⟨ Globals in the outer block 12 ⟩ +≡
font_info: array [font_index] of memory_word; { the font metric data }
fmem_ptr: font_index; { first unused word of font_info }
font_check: array [internal_font_number] of four_quarters; { check sum }
font_size: array [internal_font_number] of scaled; { "at" size }
font_dsize: array [internal_font_number] of scaled; { "design" size }
font_bc: array [internal_font_number] of eight_bits; { beginning (smallest) character code }
font_ec: array [internal_font_number] of eight_bits; { ending (largest) character code }
char_base: array [internal_font_number] of integer; { base addresses for char_info }
width_base: array [internal_font_number] of integer; { base addresses for widths }
height_base: array [internal_font_number] of integer; { base addresses for heights }
depth_base: array [internal_font_number] of integer; { base addresses for depths }
italic_base: array [internal_font_number] of integer; { base addresses for italic corrections }
lig_kern_base: array [internal_font_number] of integer; { base addresses for ligature/kerning programs }
kern_base: array [internal_font_number] of integer; { base addresses for kerns }
exten_base: array [internal_font_number] of integer; { base addresses for extensible recipes }
param_base: array [internal_font_number] of integer; { base addresses for font parameters }
bchar_label: array [internal_font_number] of font_index;
    { start of lig_kern program for left boundary character, non_address if there is none }
font_bchar: array [internal_font_number] of min_quarterword .. non_char;
    { right boundary character, non_char if there is none }
```

54.
```
⟨ Set initial values 13 ⟩ +≡
  fmem_ptr ← 0;
```

55. Of course we want to define macros that suppress the detail of how font information is actually packed, so that we don't have to write things like

$$font\_info[width\_base[f] + font\_info[char\_base[f] + c].qqqq.b0].sc$$

too often. The WEB definitions here make $char\_info(f)(c)$ the *four_quarters* word of font information corresponding to character c of font f. If q is such a word, $char\_width(f)(q)$ will be the character's width; hence the long formula above is at least abbreviated to

$$char\_width(f)(char\_info(f)(c)).$$

In practice we will try to fetch q first and look at several of its fields at the same time.

The italic correction of a character will be denoted by $char\_italic(f)(q)$, so it is analogous to $char\_width$. But we will get at the height and depth in a slightly different way, since we usually want to compute both height and depth if we want either one. The value of $height\_depth(q)$ will be the 8-bit quantity

$$b = height\_index \times 16 + depth\_index,$$

and if b is such a byte we will write $char\_height(f)(b)$ and $char\_depth(f)(b)$ for the height and depth of the character c for which $q = char\_info(f)(c)$. Got that?

The tag field will be called $char\_tag(q)$; and the remainder byte will be called $rem\_byte(q)$.

define *char_info_end*(#) ≡ #] .*qqqq*
define *char_info*(#) ≡ *font_info* [*char_base*[#] + *char_info_end*
define *char_width_end*(#) ≡ #.*b0*] .*sc*
define *char_width*(#) ≡ *font_info* [*width_base*[#] + *char_width_end*
define *char_exists*(#) ≡ (#.*b0* > *min_quarterword*)
define *char_italic_end*(#) ≡ (*qo*(#.*b2*)) **div** 4] .*sc*
define *char_italic*(#) ≡ *font_info* [*italic_base*[#] + *char_italic_end*
define *height_depth*(#) ≡ *qo*(#.*b1*)
define *char_height_end*(#) ≡ (#) **div** 16] .*sc*
define *char_height*(#) ≡ *font_info* [*height_base*[#] + *char_height_end*
define *char_depth_end*(#) ≡ # **mod** 16] .*sc*
define *char_depth*(#) ≡ *font_info* [*depth_base*[#] + *char_depth_end*
define *char_tag*(#) ≡ ((*qo*(#.*b2*)) **mod** 4)
define *skip_byte*(#) ≡ *qo*(#.*b0*)
define *next_char*(#) ≡ #.*b1*
define *op_byte*(#) ≡ *qo*(#.*b2*)
define *rem_byte*(#) ≡ #.*b3*

56. Here are some macros that help process ligatures and kerns. We write $char\_kern(f)(j)$ to find the amount of kerning specified by kerning command j in font f.

define *lig_kern_start*(#) ≡ *lig_kern_base*[#] + *rem_byte* { beginning of lig/kern program }
define *lig_kern_restart_end*(#) ≡ 256 ∗ (*op_byte*(#)) + *rem_byte*(#)
define *lig_kern_restart*(#) ≡ *lig_kern_base*[#] + *lig_kern_restart_end*
define *char_kern_end*(#) ≡ 256 ∗ (*op_byte*(#) − 128) + *rem_byte*(#)] .*sc*
define *char_kern*(#) ≡ *font_info* [*kern_base*[#] + *char_kern_end*

57. Font parameters are referred to as *slant*(*f*), *space*(*f*), etc.

define *param_end*(#) ≡ *param_base*[#]] .*sc*
define *param*(#) ≡ *font_info* [# + *param_end*
define *slant* ≡ *param*(1) { slant to the right, per unit distance upward }
define *space* ≡ *param*(2) { normal space between words }
define *x_height* ≡ *param*(5) { one ex }
define *default_rule_thickness* ≡ *param*(8) { thickness of rules }

58. Here is the subroutine that inputs the information on *tfm_file*, assuming that the file has just been reset. Parameter *f* tells which metric file is being read (either *title_font* or *label_font* or *gray_font* or *slant_font* or *logo_font*); parameter *s* is the "at" size, which will be substituted for the design size if it is positive.

This routine does only limited checking of the validity of the file, because another program (`TFtoPL`) is available to diagnose errors in the rare case that something is amiss.

define *bad_tfm* = 11 { label for *read_font_info* }
define *abend* ≡ **goto** *bad_tfm* { do this when the `TFM` data is wrong }

procedure *read_font_info*(*f* : *integer*; *s* : *scaled*); { input a `TFM` file }
label *done*, *bad_tfm*;
var *k*: *font_index*; { index into *font_info* }
lf, *lh*, *bc*, *ec*, *nw*, *nh*, *nd*, *ni*, *nl*, *nk*, *ne*, *np*: 0 .. 65535; { sizes of subfiles }
bch_label: *integer*; { left boundary label for ligatures }
bchar: 0 .. 256; { right boundary character for ligatures }
qw: *four_quarters*; *sw*: *scaled*; { accumulators }
z: *scaled*; { the design size or the "at" size }
alpha: *integer*; *beta*: 1 .. 16; { auxiliary quantities used in fixed-point multiplication }
begin ⟨ Read and check the font data; *abend* if the `TFM` file is malformed; otherwise **goto** *done* 59 ⟩;
bad_tfm: *print_nl*(´`Bad␣TFM␣file␣for`´);
case *f* **of**
title_font: *abort*(´`titles!`´);
label_font: *abort*(´`labels!`´);
gray_font: *abort*(´`pixels!`´);
slant_font: *abort*(´`slants!`´);
logo_font: *abort*(´`METAFONT␣logo!`´);
end; { there are no other cases }
done: { it might be good to close *tfm_file* now }
end;

59. ⟨ Read and check the font data; *abend* if the `TFM` file is malformed; otherwise **goto** *done* 59 ⟩ ≡
⟨ Read the `TFM` size fields 60 ⟩;
⟨ Use size fields to allocate font information 61 ⟩;
⟨ Read the `TFM` header 62 ⟩;
⟨ Read character data 63 ⟩;
⟨ Read box dimensions 64 ⟩;
⟨ Read ligature/kern program 66 ⟩;
⟨ Read extensible character recipes 67 ⟩;
⟨ Read font parameters 68 ⟩;
⟨ Make final adjustments and **goto** *done* 69 ⟩

This code is used in section 58.

60. **define** *read_two_halves_end*(#) ≡ # ← *b2* ∗ 256 + *b3*
define *read_two_halves*(#) ≡ *read_tfm_word*; # ← *b0* ∗ 256 + *b1*; *read_two_halves_end*

⟨ Read the TFM size fields 60 ⟩ ≡
begin *read_two_halves*(*lf*)(*lh*); *read_two_halves*(*bc*)(*ec*);
if (*bc* > *ec* + 1) ∨ (*ec* > 255) **then** *abend*;
if *bc* > 255 **then** { *bc* = 256 and *ec* = 255 }
begin *bc* ← 1; *ec* ← 0;
end;
read_two_halves(*nw*)(*nh*); *read_two_halves*(*nd*)(*ni*); *read_two_halves*(*nl*)(*nk*); *read_two_halves*(*ne*)(*np*);
if *lf* ≠ 6 + *lh* + (*ec* − *bc* + 1) + *nw* + *nh* + *nd* + *ni* + *nl* + *nk* + *ne* + *np* **then** *abend*;
end

This code is used in section 59.

61. The preliminary settings of the index variables *width_base*, *lig_kern_base*, *kern_base*, and *exten_base* will be corrected later by subtracting *min_quarterword* from them; and we will subtract 1 from *param_base* too. It's best to forget about such anomalies until later.

⟨ Use size fields to allocate font information 61 ⟩ ≡
lf ← *lf* − 6 − *lh*; { *lf* words should be loaded into *font_info* }
if *np* < 8 **then** *lf* ← *lf* + 8 − *np*; { at least eight parameters will appear }
if *fmem_ptr* + *lf* > *font_mem_size* **then** *abort*(´No␣room␣for␣TFM␣file!´);
char_base[*f*] ← *fmem_ptr* − *bc*; *width_base*[*f*] ← *char_base*[*f*] + *ec* + 1;
height_base[*f*] ← *width_base*[*f*] + *nw*; *depth_base*[*f*] ← *height_base*[*f*] + *nh*;
italic_base[*f*] ← *depth_base*[*f*] + *nd*; *lig_kern_base*[*f*] ← *italic_base*[*f*] + *ni*;
kern_base[*f*] ← *lig_kern_base*[*f*] + *nl*; *exten_base*[*f*] ← *kern_base*[*f*] + *nk*;
param_base[*f*] ← *exten_base*[*f*] + *ne*

This code is used in section 59.

62. Only the first two words of the header are needed by GFtoDVI.

define *store_four_quarters*(#) ≡
begin *read_tfm_word*; *qw.b0* ← *qi*(*b0*); *qw.b1* ← *qi*(*b1*); *qw.b2* ← *qi*(*b2*); *qw.b3* ← *qi*(*b3*);
← *qw*;
end

⟨ Read the TFM header 62 ⟩ ≡
begin if *lh* < 2 **then** *abend*;
store_four_quarters(*font_check*[*f*]); *read_tfm_word*;
if *b0* > 127 **then** *abend*; { design size must be positive }
z ← ((*b0* ∗ 256 + *b1*) ∗ 256 + *b2*) ∗ 16 + (*b3* **div** 16);
if *z* < *unity* **then** *abend*;
while *lh* > 2 **do**
begin *read_tfm_word*; *decr*(*lh*); { ignore the rest of the header }
end;
font_dsize[*f*] ← *z*;
if *s* > 0 **then** *z* ← *s*;
font_size[*f*] ← *z*;
end

This code is used in section 59.

63. ⟨ Read character data 63 ⟩ ≡

```
for k ← fmem_ptr to width_base[f] − 1 do
  begin store_four_quarters(font_info[k].qqqq);
  if (b0 ≥ nw) ∨ (b1 div ´20 ≥ nh) ∨ (b1 mod ´20 ≥ nd) ∨ (b2 div 4 ≥ ni) then abend;
  case b2 mod 4 of
  lig_tag: if b3 ≥ nl then abend;
  ext_tag: if b3 ≥ ne then abend;
  no_tag, list_tag: do_nothing;
  end; { there are no other cases }
  end
```

This code is used in section 59.

64. A *fix_word* whose four bytes are $(b0, b1, b2, b3)$ from left to right represents the number

$$x = \begin{cases} b_1 \cdot 2^{-4} + b_2 \cdot 2^{-12} + b_3 \cdot 2^{-20}, & \text{if } b_0 = 0; \\ -16 + b_1 \cdot 2^{-4} + b_2 \cdot 2^{-12} + b_3 \cdot 2^{-20}, & \text{if } b_0 = 255. \end{cases}$$

(No other choices of *b0* are allowed, since the magnitude of a number in design-size units must be less than 16.) We want to multiply this quantity by the integer z, which is known to be less than 2^{27}. Let $\alpha = 16z$. If $z < 2^{23}$, the individual multiplications $b \cdot z$, $c \cdot z$, $d \cdot z$ cannot overflow; otherwise we will divide z by 2, 4, 8, or 16, to obtain a multiplier less than 2^{23}, and we can compensate for this later. If z has thereby been replaced by $z' = z/2^e$, let $\beta = 2^{4-e}$; we shall compute

$$\lfloor (b_1 + b_2 \cdot 2^{-8} + b_3 \cdot 2^{-16})\, z'/\beta \rfloor$$

if $a = 0$, or the same quantity minus α if $a = 255$.

```
define store_scaled(#) ≡
        begin read_tfm_word; sw ← (((((b3 ∗ z) div ´400) + (b2 ∗ z)) div ´400) + (b1 ∗ z)) div beta;
        if b0 = 0 then # ← sw else if b0 = 255 then # ← sw − alpha else abend;
        end
```

⟨ Read box dimensions 64 ⟩ ≡

```
begin ⟨ Replace z by z′ and compute α, β 65 ⟩;
for k ← width_base[f] to lig_kern_base[f] − 1 do store_scaled(font_info[k].sc);
if font_info[width_base[f]].sc ≠ 0 then abend; { width[0] must be zero }
if font_info[height_base[f]].sc ≠ 0 then abend; { height[0] must be zero }
if font_info[depth_base[f]].sc ≠ 0 then abend; { depth[0] must be zero }
if font_info[italic_base[f]].sc ≠ 0 then abend; { italic[0] must be zero }
end
```

This code is used in section 59.

65. ⟨ Replace z by z' and compute α, β 65 ⟩ ≡

```
begin alpha ← 16 ∗ z; beta ← 16;
while z ≥ ´40000000 do
  begin z ← z div 2; beta ← beta div 2;
  end;
end
```

This code is used in section 64.

66. **define** *check_byte_range*(#) ≡
begin if (# < *bc*) ∨ (# > *ec*) **then** *abend*
end

⟨ Read ligature/kern program 66 ⟩ ≡
```
begin bch_label ← ´77777; bchar ← 256;
if nl > 0 then
  begin for k ← lig_kern_base[f] to kern_base[f] − 1 do
    begin store_four_quarters(font_info[k].qqqq);
    if b0 > stop_flag then
      begin if 256 ∗ b2 + b3 ≥ nl then abend;
      if b0 = 255 then
        if k = lig_kern_base[f] then bchar ← b1;
      end
    else begin if b1 ≠ bchar then check_byte_range(b1);
      if b2 < kern_flag then check_byte_range(b3)
      else if 256 ∗ (b2 − 128) + b3 ≥ nk then abend;
      end;
    end;
  if b0 = 255 then bch_label ← 256 ∗ b2 + b3;
  end;
for k ← kern_base[f] to exten_base[f] − 1 do store_scaled(font_info[k].sc);
end
```
This code is used in section 59.

67. ⟨ Read extensible character recipes 67 ⟩ ≡
```
for k ← exten_base[f] to param_base[f] − 1 do
  begin store_four_quarters(font_info[k].qqqq);
  if b0 ≠ 0 then check_byte_range(b0);
  if b1 ≠ 0 then check_byte_range(b1);
  if b2 ≠ 0 then check_byte_range(b2);
  check_byte_range(b3);
  end
```
This code is used in section 59.

68. ⟨ Read font parameters 68 ⟩ ≡
```
begin for k ← 1 to np do
  if k = 1 then { the slant parameter is a pure number }
    begin read_tfm_word;
    if b0 > 127 then sw ← b0 − 256 else sw ← b0;
    sw ← sw ∗ ´400 + b1; sw ← sw ∗ ´400 + b2; font_info[param_base[f]].sc ← (sw ∗ ´20) + (b3 div ´20);
    end
  else store_scaled(font_info[param_base[f] + k − 1].sc);
for k ← np + 1 to 8 do font_info[param_base[f] + k − 1].sc ← 0;
end
```
This code is used in section 59.

69. Now to wrap it up, we have checked all the necessary things about the `TFM` file, and all we need to do is put the finishing touches on the data for the new font.

define *adjust*(#) ≡ #[*f*] ← *qo*(#[*f*]) { correct for the excess *min_quarterword* that was added }

⟨ Make final adjustments and **goto** *done* 69 ⟩ ≡
font_bc[*f*] ← *bc*; *font_ec*[*f*] ← *ec*;
if *bch_label* < *nl* **then** *bchar_label*[*f*] ← *bch_label* + *lig_kern_base*[*f*]
else *bchar_label*[*f*] ← *non_address*;
font_bchar[*f*] ← *qi*(*bchar*); *adjust*(*width_base*); *adjust*(*lig_kern_base*); *adjust*(*kern_base*);
adjust(*exten_base*); *decr*(*param_base*[*f*]); *fmem_ptr* ← *fmem_ptr* + *lf*; **goto** *done*

This code is used in section 59.

70. The string pool. GFtoDVI remembers strings by putting them into an array called *str_pool*. The *str_start* array tells where each string starts in the pool.

⟨ Types in the outer block 9 ⟩ +≡
  pool_pointer = 0 .. *pool_size*; { for variables that point into *str_pool* }
  str_number = 0 .. *max_strings*; { for variables that point into *str_start* }

71. As new strings enter, we keep track of the storage currently used, by means of two global variables called *pool_ptr* and *str_ptr*. These are periodically reset to their initial values when we move from one character to another, because most strings are of only temporary interest.

⟨ Globals in the outer block 12 ⟩ +≡
str_pool: **packed array** [*pool_pointer*] **of** *ASCII_code*; { the characters }
str_start: **array** [*str_number*] **of** *pool_pointer*; { the starting pointers }
pool_ptr: *pool_pointer*; { first unused position in *str_pool* }
str_ptr: *str_number*; { start of the current string being created }
init_str_ptr: *str_number*; { *str_ptr* setting when a new character starts }

72. Several of the elementary string operations are performed using WEB macros instead of using Pascal procedures, because many of the operations are done quite frequently and we want to avoid the overhead of procedure calls. For example, here is a simple macro that computes the length of a string.

define *length*(#) ≡ (*str_start*[# + 1] − *str_start*[#]) { the number of characters in string number # }

73. Strings are created by appending character codes to *str_pool*. The macro called *append_char*, defined here, does not check to see if the value of *pool_ptr* has gotten too high; that test is supposed to be made before *append_char* is used.

To test if there is room to append *l* more characters to *str_pool*, we shall write *str_room*(*l*), which aborts GFtoDVI and gives an apologetic error message if there isn't enough room.

define *append_char*(#) ≡ { put *ASCII_code* # at the end of *str_pool* }
  begin *str_pool*[*pool_ptr*] ← #; *incr*(*pool_ptr*);
  end
define *str_room*(#) ≡ { make sure that the pool hasn't overflowed }
  begin if *pool_ptr* + # > *pool_size* **then** *abort*(´Too␣many␣strings!´);
  end

74. Once a sequence of characters has been appended to *str_pool*, it officially becomes a string when the function *make_string* is called. This function returns the identification number of the new string as its value.

function *make_string*: *str_number*; { current string enters the pool }
  begin if *str_ptr* = *max_strings* **then** *abort*(´Too␣many␣labels!´);
  incr(*str_ptr*); *str_start*[*str_ptr*] ← *pool_ptr*; *make_string* ← *str_ptr* − 1;
  end;

75. The first strings in the string pool are the keywords that `GFtoDVI` recognizes in the *xxx* commands of a `GF` file. They are entered into *str_pool* by means of a tedious bunch of assignment statements, together with calls on the *first_string* subroutine.

```
define init_str0(#) ≡ first_string(#)
define init_str1(#) ≡ buffer[1] ← #; init_str0
define init_str2(#) ≡ buffer[2] ← #; init_str1
define init_str3(#) ≡ buffer[3] ← #; init_str2
define init_str4(#) ≡ buffer[4] ← #; init_str3
define init_str5(#) ≡ buffer[5] ← #; init_str4
define init_str6(#) ≡ buffer[6] ← #; init_str5
define init_str7(#) ≡ buffer[7] ← #; init_str6
define init_str8(#) ≡ buffer[8] ← #; init_str7
define init_str9(#) ≡ buffer[9] ← #; init_str8
define init_str10(#) ≡ buffer[10] ← #; init_str9
define init_str11(#) ≡ buffer[11] ← #; init_str10
define init_str12(#) ≡ buffer[12] ← #; init_str11
define init_str13(#) ≡ buffer[13] ← #; init_str12
define longest_keyword = 13
procedure first_string(c : integer);
  begin if str_ptr ≠ c then abort('?'); { internal consistency check }
  while l > 0 do
    begin append_char(buffer[l]); decr(l);
    end;
  incr(str_ptr); str_start[str_ptr] ← pool_ptr;
  end;
```

76. ⟨ Globals in the outer block 12 ⟩ +≡

```
l: integer; { length of string being made by first_string }
```

77. Here are the tedious assignments just promised. String number 0 is the empty string.

define *null_string* = 0 { the empty keyword }
define *area_code* = 4 { add to font code for the 'area' keywords }
define *at_code* = 8 { add to font code for the 'at' keywords }
define *rule_code* = 13 { code for the keyword 'rule' }
define *title_code* = 14 { code for the keyword 'title' }
define *rule_thickness_code* = 15 { code for the keyword 'rulethickness' }
define *offset_code* = 16 { code for the keyword 'offset' }
define *x_offset_code* = 17 { code for the keyword 'xoffset' }
define *y_offset_code* = 18 { code for the keyword 'yoffset' }
define *max_keyword* = 18 { largest keyword code number }

⟨ Initialize the strings 77 ⟩ ≡
str_ptr ← 0; *pool_ptr* ← 0; *str_start*[0] ← 0;
l ← 0; *init_str0*(*null_string*);
l ← 9; *init_str9*("t")("i")("t")("l")("e")("f")("o")("n")("t")(*title_font*);
l ← 9; *init_str9*("l")("a")("b")("e")("l")("f")("o")("n")("t")(*label_font*);
l ← 8; *init_str8*("g")("r")("a")("y")("f")("o")("n")("t")(*gray_font*);
l ← 9; *init_str9*("s")("l")("a")("n")("t")("f")("o")("n")("t")(*slant_font*);
l ← 13;
init_str13("t")("i")("t")("l")("e")("f")("o")("n")("t")("a")("r")("e")("a")(*title_font* + *area_code*);
l ← 13;
init_str13("l")("a")("b")("e")("l")("f")("o")("n")("t")("a")("r")("e")("a")(*label_font* + *area_code*);
l ← 12;
init_str12("g")("r")("a")("y")("f")("o")("n")("t")("a")("r")("e")("a")(*gray_font* + *area_code*);
l ← 13;
init_str13("s")("l")("a")("n")("t")("f")("o")("n")("t")("a")("r")("e")("a")(*slant_font* + *area_code*);
l ← 11; *init_str11*("t")("i")("t")("l")("e")("f")("o")("n")("t")("a")("t")(*title_font* + *at_code*);
l ← 11; *init_str11*("l")("a")("b")("e")("l")("f")("o")("n")("t")("a")("t")(*label_font* + *at_code*);
l ← 10; *init_str10*("g")("r")("a")("y")("f")("o")("n")("t")("a")("t")(*gray_font* + *at_code*);
l ← 11; *init_str11*("s")("l")("a")("n")("t")("f")("o")("n")("t")("a")("t")(*slant_font* + *at_code*);
l ← 4; *init_str4*("r")("u")("l")("e")(*rule_code*);
l ← 5; *init_str5*("t")("i")("t")("l")("e")(*title_code*);
l ← 13;
init_str13("r")("u")("l")("e")("t")("h")("i")("c")("k")("n")("e")("s")("s")(*rule_thickness_code*);
l ← 6; *init_str6*("o")("f")("f")("s")("e")("t")(*offset_code*);
l ← 7; *init_str7*("x")("o")("f")("f")("s")("e")("t")(*x_offset_code*);
l ← 7; *init_str7*("y")("o")("f")("f")("s")("e")("t")(*y_offset_code*);

See also sections 78 and 88.

This code is used in section 219.

78. We will also find it useful to have the following strings. (The names of default fonts will presumably be different at different sites.)

define *gf_ext* = *max_keyword* + 1 { string number for '.gf' }
define *dvi_ext* = *max_keyword* + 2 { string number for '.dvi' }
define *tfm_ext* = *max_keyword* + 3 { string number for '.tfm' }
define *page_header* = *max_keyword* + 4 { string number for '␣␣Page␣' }
define *char_header* = *max_keyword* + 5 { string number for '␣␣Character␣' }
define *ext_header* = *max_keyword* + 6 { string number for '␣␣Ext␣' }
define *left_quotes* = *max_keyword* + 7 { string number for '␣␣``' }
define *right_quotes* = *max_keyword* + 8 { string number for '''' }
define *equals_sign* = *max_keyword* + 9 { string number for ' = ' }
define *plus_sign* = *max_keyword* + 10 { string number for ' + (' }
define *default_title_font* = *max_keyword* + 11 { string number for the default *title_font* }
define *default_label_font* = *max_keyword* + 12 { string number for the default *label_font* }
define *default_gray_font* = *max_keyword* + 13 { string number for the default *gray_font* }
define *logo_font_name* = *max_keyword* + 14 { string number for the font with METAFONT logo }
define *small_logo* = *max_keyword* + 15 { string number for 'METAFONT' }
define *home_font_area* = *max_keyword* + 16 { string number for system-dependent font area }

⟨ Initialize the strings 77 ⟩ +≡
$l \leftarrow 3$; *init_str3*(".")("g")("f")(*gf_ext*);
$l \leftarrow 4$; *init_str4*(".")("d")("v")("i")(*dvi_ext*);
$l \leftarrow 4$; *init_str4*(".")("t")("f")("m")(*tfm_ext*);
$l \leftarrow 7$; *init_str7*("␣")("␣")("P")("a")("g")("e")("␣")(*page_header*);
$l \leftarrow 12$; *init_str12*("␣")("␣")("C")("h")("a")("r")("a")("c")("t")("e")("r")("␣")(*char_header*);
$l \leftarrow 6$; *init_str6*("␣")("␣")("E")("x")("t")("␣")(*ext_header*);
$l \leftarrow 4$; *init_str4*("␣")("␣")("`")("`")(*left_quotes*);
$l \leftarrow 2$; *init_str2*("´")("´")(*right_quotes*);
$l \leftarrow 3$; *init_str3*("␣")("=")("␣")(*equals_sign*);
$l \leftarrow 4$; *init_str4*("␣")("+")("␣")("(")(*plus_sign*);
$l \leftarrow 4$; *init_str4*("c")("m")("r")("8")(*default_title_font*);
$l \leftarrow 6$; *init_str6*("c")("m")("t")("t")("1")("0")(*default_label_font*);
$l \leftarrow 4$; *init_str4*("g")("r")("a")("y")(*default_gray_font*);
$l \leftarrow 5$; *init_str5*("l")("o")("g")("o")("8")(*logo_font_name*);
$l \leftarrow 8$; *init_str8*("M")("E")("T")("A")("F")("O")("N")("T")(*small_logo*);

79. If an *xxx* command has just been encountered in the GF file, the following procedure interprets its keyword. More precisely, we assume that *cur_gf* contains an op-code byte just read from the GF file, where *xxx1* ≤ *cur_gf* ≤ *no_op*. The *interpret_xxx* procedure will read the rest of the command, in the following way:

1) If *cur_gf* is *no_op* or *yyy*, or if it's an *xxx* command with an unknown keyword, the bytes are simply read and ignored, and the value *no_operation* is returned.
2) If *cur_gf* is an *xxx* command (either *xxx1* or ··· or *xxx4*), and if the associated string matches a keyword exactly, the string number of that keyword is returned (e.g., *rule_thickness_code*).
3) If *cur_gf* is an *xxx* command whose string begins with keyword and space, the string number of that keyword is returned, and the remainder of the string is put into the string pool (where it will be string number *cur_string*. Exception: If the keyword is *null_string*, the character immediately following the blank space is put into the global variable *label_type*, and the remaining characters go into the string pool.

In all cases, *cur_gf* will then be reset to the op-code byte that immediately follows the original command.

```
define no_operation = max_keyword + 1
⟨ Types in the outer block 9 ⟩ +≡
  keyword_code = null_string .. no_operation;
```

80.
```
⟨ Globals in the outer block 12 ⟩ +≡
cur_gf: eight_bits;  { the byte most recently read from gf_file }
cur_string: str_number;  { the string following a keyword and space }
label_type: eight_bits;  { the character following a null keyword and space }
```

81. We will be using this procedure when reading the GF file just after the preamble and just after *eoc* commands.

```
function interpret_xxx: keyword_code;
  label done, done1, not_found;
  var k: integer;  { number of bytes in an xxx command }
    j: integer;  { number of bytes read so far }
    l: 0 .. longest_keyword;  { length of keyword to check }
    m: keyword_code;  { runs through the list of known keywords }
    n1: 0 .. longest_keyword;  { buffered character being checked }
    n2: pool_pointer;  { pool character being checked }
    c: keyword_code;  { the result to return }
  begin c ← no_operation; cur_string ← null_string;
  case cur_gf of
  no_op: goto done;
  yyy: begin k ← signed_quad; goto done;
    end;
  xxx1: k ← get_byte;
  xxx2: k ← get_two_bytes;
  xxx3: k ← get_three_bytes;
  xxx4: k ← signed_quad;
  end;  { there are no other cases }
  ⟨ Read the next k characters of the GF file; change c and goto done if a keyword is recognized 82 ⟩;
done: cur_gf ← get_byte; interpret_xxx ← c;
  end;
```

82. ⟨ Read the next k characters of the `GF` file; change c and **goto** *done* if a keyword is recognized 82 ⟩ ≡

```
j ← 0; if k < 2 then goto not_found;
loop begin l ← j;
  if j = k then goto done1;
  if j = longest_keyword then goto not_found;
  incr(j); buffer[j] ← get_byte;
  if buffer[j] = "␣" then goto done1;
  end;
done1: ⟨ If the keyword in buffer[1 .. l] is known, change c and goto done 83 ⟩;
not_found: while j < k do
  begin incr(j); cur_gf ← get_byte;
  end
```

This code is used in section 81.

83. ⟨ If the keyword in *buffer*[1 .. *l*] is known, change c and **goto** *done* 83 ⟩ ≡

```
for m ← null_string to max_keyword do
  if length(m) = l then
    begin n1 ← 0; n2 ← str_start[m];
    while (n1 < l) ∧ (buffer[n1 + 1] = str_pool[n2]) do
      begin incr(n1); incr(n2);
      end;
    if n1 = l then
      begin c ← m;
      if m = null_string then
        begin incr(j); label_type ← get_byte;
        end;
      str_room(k − j);
      while j < k do
        begin incr(j); append_char(get_byte);
        end;
      cur_string ← make_string; goto done;
      end;
    end
```

This code is used in section 82.

84. When an *xxx* command takes a numeric argument, *get_yyy* reads that argument and puts the following byte into *cur_gf*.

```
function get_yyy: scaled;
  var v: scaled;  { value just read }
  begin if cur_gf ≠ yyy then get_yyy ← 0
  else begin v ← signed_quad; cur_gf ← get_byte; get_yyy ← v;
    end;
  end;
```

85. A simpler method is used for special commands between *boc* and *eoc*, since `GFtoDVI` doesn't even look at them.

```
procedure skip_nop;
  label done;
  var k: integer;  { number of bytes in an xxx command }
    j: integer;  { number of bytes read so far }
  begin case cur_gf of
  no_op: goto done;
  yyy: begin k ← signed_quad; goto done;
    end;
  xxx1: k ← get_byte;
  xxx2: k ← get_two_bytes;
  xxx3: k ← get_three_bytes;
  xxx4: k ← signed_quad;
  end;  { there are no other cases }
  for j ← 1 to k do cur_gf ← get_byte;
done: cur_gf ← get_byte;
  end;
```

86. File names. It's time now to fret about file names. `GFtoDVI` uses the conventions of TEX and METAFONT to convert file names into strings that can be used to open files. Three routines called *begin_name*, *more_name*, and *end_name* are involved, so that the system-dependent parts of file naming conventions are isolated from the system-independent ways in which file names are used. (See the TEX or METAFONT program listing for further explanation.)

```
⟨ Globals in the outer block 12 ⟩ +≡
cur_name: str_number;   { name of file just scanned }
cur_area: str_number;   { file area just scanned, or null_string }
cur_ext: str_number;   { file extension just scanned, or null_string }
```

87. The file names we shall deal with for illustrative purposes have the following structure: If the name contains '`>`' or '`:`', the file area consists of all characters up to and including the final such character; otherwise the file area is null. If the remaining file name contains '`.`', the file extension consists of all such characters from the first remaining '`.`' to the end, otherwise the file extension is null.

We can scan such file names easily by using two global variables that keep track of the occurrences of area and extension delimiters:

```
⟨ Globals in the outer block 12 ⟩ +≡
area_delimiter: pool_pointer;   { the most recent '>' or ':', if any }
ext_delimiter: pool_pointer;   { the relevant '.', if any }
```

88. Font metric files whose areas are not given explicitly are assumed to appear in a standard system area called *home_font_area*. This system area name will, of course, vary from place to place. The program here sets it to '`TeXfonts:`'.

```
⟨ Initialize the strings 77 ⟩ +≡
  l ← 9; init_str9("T")("e")("X")("f")("o")("n")("t")("s")(":")(home_font_area);
```

89. Here now is the first of the system-dependent routines for file name scanning.

```
procedure begin_name;
  begin area_delimiter ← 0; ext_delimiter ← 0;
  end;
```

90. And here's the second.

```
function more_name(c : ASCII_code): boolean;
  begin if c = " " then more_name ← false
  else begin if (c = ">") ∨ (c = ":") then
      begin area_delimiter ← pool_ptr; ext_delimiter ← 0;
      end
    else if (c = ".") ∧ (ext_delimiter = 0) then ext_delimiter ← pool_ptr;
    str_room(1); append_char(c);   { contribute c to the current string }
    more_name ← true;
    end;
  end;
```

91. The third.

```
procedure end_name;
  begin if str_ptr + 3 > max_strings then abort('Too many strings!');
  if area_delimiter = 0 then cur_area ← null_string
  else begin cur_area ← str_ptr; incr(str_ptr); str_start[str_ptr] ← area_delimiter + 1;
    end;
  if ext_delimiter = 0 then
    begin cur_ext ← null_string; cur_name ← make_string;
    end
  else begin cur_name ← str_ptr; incr(str_ptr); str_start[str_ptr] ← ext_delimiter;
    cur_ext ← make_string;
    end;
  end;
```

92. Another system-dependent routine is needed to convert three strings into the *name_of_file* value that is used to open files. The present code allows both lowercase and uppercase letters in the file name.

```
  define append_to_name(#) ≡
           begin c ← #; incr(k);
           if k ≤ file_name_size then name_of_file[k] ← xchr[c];
           end
procedure pack_file_name(n, a, e : str_number);
  var k: integer; { number of positions filled in name_of_file }
    c: ASCII_code; { character being packed }
    j: integer; { index into str_pool }
    name_length: 0 .. file_name_size; { number of characters packed }
  begin k ← 0;
  for j ← str_start[a] to str_start[a + 1] − 1 do append_to_name(str_pool[j]);
  for j ← str_start[n] to str_start[n + 1] − 1 do append_to_name(str_pool[j]);
  for j ← str_start[e] to str_start[e + 1] − 1 do append_to_name(str_pool[j]);
  if k ≤ file_name_size then name_length ← k else name_length ← file_name_size;
  for k ← name_length + 1 to file_name_size do name_of_file[k] ← ' ';
  end;
```

93. Now let's consider the routines by which GFtoDVI deals with file names in a system-independent manner. The global variable *job_name* contains the GF file name that is being input. This name is extended by '.dvi' in order to make the name of the output file.

```
⟨ Globals in the outer block 12 ⟩ +≡
job_name: str_number; { principal file name }
```

94. The *start_gf* procedure prompts the user for the name of the generic font file to be input. It opens the file, making sure that some input is present; then it opens the output file.

Although this routine is system-independent, it should probably be modified to take the file name from the command line (without an initial prompt), on systems that permit such things.

```
procedure start_gf;
  label found, done;
  begin loop begin print_nl('GF␣file␣name:␣'); input_ln; buf_ptr ← 0; buffer[line_length] ← "?";
    while buffer[buf_ptr] = "␣" do incr(buf_ptr);
    if buf_ptr < line_length then
      begin ⟨ Scan the file name in the buffer 95 ⟩;
      if cur_ext = null_string then cur_ext ← gf_ext;
      pack_file_name(cur_name, cur_area, cur_ext); open_gf_file;
      if ¬eof(gf_file) then goto found;
      print_nl('Oops...␣I␣can't␣find␣file␣'); print(name_of_file);
      end;
    end;
found: job_name ← cur_name; pack_file_name(job_name, null_string, dvi_ext); open_dvi_file;
  end;
```

95. ⟨ Scan the file name in the buffer 95 ⟩ ≡

```
  if buffer[line_length − 1] = "/" then
    begin interaction ← true; decr(line_length);
    end;
  begin_name;
  loop begin if buf_ptr = line_length then goto done;
    if ¬more_name(buffer[buf_ptr]) then goto done;
    incr(buf_ptr);
    end;
done: end_name
```

This code is used in section 94.

96. Special instructions found near the beginning of the GF file might change the names, areas, and "at" sizes of the fonts that GFtoDVI will be using. But when we reach the first *boc* instruction, we input all of the TFM files. The global variable *interaction* is set *true* if a "/" was removed at the end of the file name; this means that the user will have a chance to issue special instructions online just before the fonts are loaded.

```
  define check_fonts ≡ if fonts_not_loaded then load_fonts
```

⟨ Globals in the outer block 12 ⟩ +≡

```
interaction: boolean;   { is the user allowed to type specials online? }
fonts_not_loaded: boolean;   { have the TFM files still not been input? }
font_name: array [internal_font_number] of str_number;   { current font names }
font_area: array [internal_font_number] of str_number;   { current font areas }
font_at: array [internal_font_number] of scaled;   { current font "at" sizes }
```

97. ⟨ Set initial values 13 ⟩ +≡

```
  interaction ← false; fonts_not_loaded ← true; font_name[title_font] ← default_title_font;
  font_name[label_font] ← default_label_font; font_name[gray_font] ← default_gray_font;
  font_name[slant_font] ← null_string; font_name[logo_font] ← logo_font_name;
  for k ← title_font to logo_font do
    begin font_area[k] ← null_string; font_at[k] ← 0;
    end;
```

98. After the following procedure has been performed, there will be no turning back; the fonts will have been firmly established in `GFtoDVI`'s memory.

```
⟨ Declare the procedure called load_fonts 98 ⟩ ≡
procedure load_fonts;
  label done, continue, found, not_found;
  var f: internal_font_number; i: four_quarters;  { font information word }
    j, k, v: integer;  { registers for initializing font tables }
    m: title_font .. slant_font + area_code;  { keyword found }
    n1: 0 .. longest_keyword;  { buffered character being checked }
    n2: pool_pointer;  { pool character being checked }
  begin if interaction then ⟨ Get online special input 99 ⟩;
  fonts_not_loaded ← false;
  for f ← title_font to logo_font do
    if (f ≠ slant_font) ∨ (length(font_name[f]) > 0) then
      begin if length(font_area[f]) = 0 then font_area[f] ← home_font_area;
      pack_file_name(font_name[f], font_area[f], tfm_ext); open_tfm_file; read_font_info(f, font_at[f]);
      if font_area[f] = home_font_area then font_area[f] ← null_string;
      dvi_font_def(f);  { put the font name in the DVI file }
      end;
  ⟨ Initialize global variables that depend on the font data 137 ⟩;
  end;
```

This code is used in section 111.

99.

```
⟨ Get online special input 99 ⟩ ≡
  loop begin not_found: print_nl('Special␣font␣substitution:␣');
  continue: input_ln;
    if line_length = 0 then goto done;
    ⟨ Search buffer for valid keyword; if successful, goto found 100 ⟩;
    print('Please␣say␣e.g.␣"grayfont␣foo"␣or␣"slantfontarea␣baz".'); goto not_found;
  found: ⟨ Update the font name or area 101 ⟩;
    print('OK;␣any␣more?␣'); goto continue;
    end;
done:
```

This code is used in section 98.

100.

```
⟨ Search buffer for valid keyword; if successful, goto found 100 ⟩ ≡
  buf_ptr ← 0; buffer[line_length] ← "␣";
  while buffer[buf_ptr] ≠ "␣" do incr(buf_ptr);
  for m ← title_font to slant_font + area_code do
    if length(m) = buf_ptr then
      begin n1 ← 0; n2 ← str_start[m];
      while (n1 < buf_ptr) ∧ (buffer[n1] = str_pool[n2]) do
        begin incr(n1); incr(n2);
        end;
      if n1 = buf_ptr then goto found;
      end
```

This code is used in section 99.

101. ⟨ Update the font name or area 101 ⟩ ≡

```
incr(buf_ptr); str_room(line_length − buf_ptr);
while buf_ptr < line_length do
  begin append_char(buffer[buf_ptr]); incr(buf_ptr);
  end;
if m > area_code then font_area[m − area_code] ← make_string
else begin font_name[m] ← make_string; font_area[m] ← null_string; font_at[m] ← 0;
  end;
init_str_ptr ← str_ptr
```

This code is used in section 99.

102. Shipping pages out. The following routines are used to write the DVI file. They have been copied from TEX, but simplified; we don't have to handle nearly as much generality as TEX does.

Statistics about the entire set of pages that will be shipped out must be reported in the DVI postamble. The global variables *total_pages*, *max_v*, *max_h*, and *last_bop* are used to record this information.

```
⟨ Globals in the outer block 12 ⟩ +≡
total_pages: integer;  { the number of pages that have been shipped out }
max_v: scaled;  { maximum height-plus-depth of pages shipped so far }
max_h: scaled;  { maximum width of pages shipped so far }
last_bop: integer;  { location of previous bop in the DVI output }
```

103.
```
⟨ Set initial values 13 ⟩ +≡
  total_pages ← 0; max_v ← 0; max_h ← 0; last_bop ← −1;
```

104. The DVI bytes are output to a buffer instead of being written directly to the output file. This makes it possible to reduce the overhead of subroutine calls.

The output buffer is divided into two parts of equal size; the bytes found in *dvi_buf*[0 .. *half_buf* − 1] constitute the first half, and those in *dvi_buf*[*half_buf* .. *dvi_buf_size* − 1] constitute the second. The global variable *dvi_ptr* points to the position that will receive the next output byte. When *dvi_ptr* reaches *dvi_limit*, which is always equal to one of the two values *half_buf* or *dvi_buf_size*, the half buffer that is about to be invaded next is sent to the output and *dvi_limit* is changed to its other value. Thus, there is always at least a half buffer's worth of information present, except at the very beginning of the job.

Bytes of the DVI file are numbered sequentially starting with 0; the next byte to be generated will be number *dvi_offset* + *dvi_ptr*.

```
⟨ Types in the outer block 9 ⟩ +≡
  dvi_index = 0 .. dvi_buf_size;  { an index into the output buffer }
```

105. Some systems may find it more efficient to make *dvi_buf* a **packed** array, since output of four bytes at once may be facilitated.

```
⟨ Globals in the outer block 12 ⟩ +≡
dvi_buf: array [dvi_index] of eight_bits;  { buffer for DVI output }
half_buf: dvi_index;  { half of dvi_buf_size }
dvi_limit: dvi_index;  { end of the current half buffer }
dvi_ptr: dvi_index;  { the next available buffer address }
dvi_offset: integer;  { dvi_buf_size times the number of times the output buffer has been fully emptied }
```

106. Initially the buffer is all in one piece; we will output half of it only after it first fills up.

```
⟨ Set initial values 13 ⟩ +≡
  half_buf ← dvi_buf_size div 2; dvi_limit ← dvi_buf_size; dvi_ptr ← 0; dvi_offset ← 0;
```

107. The actual output of *dvi_buf*[*a* .. *b*] to *dvi_file* is performed by calling *write_dvi*(*a*, *b*). It is safe to assume that *a* and *b* + 1 will both be multiples of 4 when *write_dvi*(*a*, *b*) is called; therefore it is possible on many machines to use efficient methods to pack four bytes per word and to output an array of words with one system call.

```
procedure write_dvi(a, b : dvi_index);
  var k: dvi_index;
  begin for k ← a to b do write(dvi_file, dvi_buf[k]);
  end;
```

108. To put a byte in the buffer without paying the cost of invoking a procedure each time, we use the macro *dvi_out*.

```
define dvi_out(#) ≡ begin dvi_buf[dvi_ptr] ← #; incr(dvi_ptr);
        if dvi_ptr = dvi_limit then dvi_swap;
        end
procedure dvi_swap; { outputs half of the buffer }
  begin if dvi_limit = dvi_buf_size then
    begin write_dvi(0, half_buf − 1); dvi_limit ← half_buf; dvi_offset ← dvi_offset + dvi_buf_size;
    dvi_ptr ← 0;
    end
  else begin write_dvi(half_buf, dvi_buf_size − 1); dvi_limit ← dvi_buf_size;
    end;
  end;
```

109. Here is how we clean out the buffer when TEX is all through; *dvi_ptr* will be a multiple of 4.

```
⟨ Empty the last bytes out of dvi_buf 109 ⟩ ≡
  if dvi_limit = half_buf then write_dvi(half_buf, dvi_buf_size − 1);
  if dvi_ptr > 0 then write_dvi(0, dvi_ptr − 1)
```

This code is used in section 115.

110. The *dvi_four* procedure outputs four bytes in two's complement notation, without risking arithmetic overflow.

```
procedure dvi_four(x : integer);
  begin if x ≥ 0 then dvi_out(x div ´100000000)
  else begin x ← x + ´10000000000; x ← x + ´10000000000; dvi_out((x div ´100000000) + 128);
    end;
  x ← x mod ´100000000; dvi_out(x div ´200000); x ← x mod ´200000; dvi_out(x div ´400);
  dvi_out(x mod ´400);
  end;
```

111. Here's a procedure that outputs a font definition.

```
define select_font(#) ≡ dvi_out(fnt_num_0 + #)  { set current font to # }
procedure dvi_font_def(f : internal_font_number);
  var k: integer; { index into str_pool }
  begin dvi_out(fnt_def1); dvi_out(f);
  dvi_out(qo(font_check[f].b0)); dvi_out(qo(font_check[f].b1)); dvi_out(qo(font_check[f].b2));
  dvi_out(qo(font_check[f].b3));
  dvi_four(font_size[f]); dvi_four(font_dsize[f]);
  dvi_out(length(font_area[f])); dvi_out(length(font_name[f]));
  ⟨ Output the font name whose internal number is f 112 ⟩;
  end;
⟨ Declare the procedure called load_fonts 98 ⟩
```

112.

```
⟨ Output the font name whose internal number is f 112 ⟩ ≡
  for k ← str_start[font_area[f]] to str_start[font_area[f] + 1] − 1 do dvi_out(str_pool[k]);
  for k ← str_start[font_name[f]] to str_start[font_name[f] + 1] − 1 do dvi_out(str_pool[k])
```

This code is used in section 111.

113. The *typeset* subroutine typesets any eight-bit character.

```
procedure typeset(c : eight_bits);
  begin if c ≥ 128 then dvi_out(set1);
  dvi_out(c);
  end;
```

114. The *dvi_scaled* subroutine takes a *real* value x and outputs a decimal approximation to $x/unity$, correct to one decimal place.

```
procedure dvi_scaled(x : real);
  var n: integer;  { an integer approximation to 10 * x/unity }
    m: integer;  { the integer part of the answer }
    k: integer;  { the number of digits in m }
  begin n ← round(x/6553.6);
  if n < 0 then
    begin dvi_out("-"); n ← −n;
    end;
  m ← n div 10; k ← 0;
  repeat incr(k); buffer[k] ← (m mod 10) + "0"; m ← m div 10;
  until m = 0;
  repeat dvi_out(buffer[k]); decr(k);
  until k = 0;
  if n mod 10 ≠ 0 then
    begin dvi_out("."); dvi_out((n mod 10) + "0");
    end;
  end;
```

115. At the end of the program, we must finish things off by writing the postamble. An integer variable k will be declared for use by this routine.

```
⟨ Finish the DVI file and goto final_end 115 ⟩ ≡
  begin dvi_out(post);  { beginning of the postamble }
  dvi_four(last_bop); last_bop ← dvi_offset + dvi_ptr − 5;  { post location }
  dvi_four(25400000); dvi_four(473628672);  { conversion ratio for sp }
  dvi_four(1000);  { magnification factor }
  dvi_four(max_v); dvi_four(max_h);
  dvi_out(0); dvi_out(3);  { 'max_push' is said to be 3 }
  dvi_out(total_pages div 256); dvi_out(total_pages mod 256);
  if ¬fonts_not_loaded then
    for k ← title_font to logo_font do
      if length(font_name[k]) > 0 then dvi_font_def(k);
  dvi_out(post_post); dvi_four(last_bop); dvi_out(dvi_id_byte);
  k ← 4 + ((dvi_buf_size − dvi_ptr) mod 4);  { the number of 223's }
  while k > 0 do
    begin dvi_out(223); decr(k);
    end;
  ⟨ Empty the last bytes out of dvi_buf 109 ⟩;
  goto final_end;
  end
```

This code is used in section 219.

116. Rudimentary typesetting. One of GFtoDVI's little duties is to be a mini-TEX: It must be able to typeset the equivalent of '\hbox{⟨string⟩}' for a given string of ASCII characters, using either the title font or the label font.

The *hbox* procedure does this. The width, height, and depth of the box defined by string s in font f are computed in global variables *box_width*, *box_height*, and *box_depth*.

The task would be trivial if it weren't for ligatures and kerns, which are implemented here in full generality. (Infinite looping is possible if the TFM file is malformed; TFtoPL will diagnose such problems.)

We assume that "␣" is a space character; character code *'40* will not be typeset unless it is accessed via a ligature.

If parameter *send_it* is *false*, we merely want to know the box dimensions. Otherwise typesetting commands are also sent to the DVI file; we assume in this case that font f has already been selected in the DVI file as the current font.

```
define set_cur_r ≡
        if k < end_k then cur_r ← qi(str_pool[k])
        else cur_r ← bchar
procedure hbox(s : str_number; f : internal_font_number; send_it : boolean);
  label continue, done;
  var k, end_k, max_k: pool_pointer;  { indices into str_pool }
    i, j: four_quarters;  { font information words }
    cur_l: 0 .. 256;  { character to the left of the "cursor" }
    cur_r: min_quarterword .. non_char;  { character to the right of the "cursor" }
    bchar: min_quarterword .. non_char;  { right boundary character }
    stack_ptr: 0 .. lig_lookahead;  { number of entries on lig_stack }
    l: font_index;  { pointer to lig/kern instruction }
    kern_amount: scaled;  { extra space to be typeset }
    hd: eight_bits;  { height and depth indices for a character }
    x: scaled;  { temporary register }
    save_c: ASCII_code;  { character temporarily blanked out }
  begin box_width ← 0; box_height ← 0; box_depth ← 0;
  k ← str_start[s]; max_k ← str_start[s + 1]; save_c ← str_pool[max_k]; str_pool[max_k] ← "␣";
  while k < max_k do
    begin if str_pool[k] = "␣" then ⟨ Typeset a space in font f and advance k 119 ⟩
    else begin end_k ← k;
      repeat incr(end_k);
      until str_pool[end_k] = "␣";
      kern_amount ← 0; cur_l ← 256; stack_ptr ← 0; bchar ← font_bchar[f]; set_cur_r;
      suppress_lig ← false;
    continue: ⟨ If there's a ligature or kern at the cursor position, update the cursor data structures,
          possibly advancing k; continue until the cursor wants to move right 120 ⟩;
      ⟨ Typeset character cur_l, if it exists in the font; also append an optional kern 121 ⟩;
      ⟨ Move the cursor to the right and goto continue, if there's more work to do in the current word 123 ⟩;
      end;  { now k = end_k }
    end;
  str_pool[max_k] ← save_c;
  end;
```

117. ⟨ Globals in the outer block 12 ⟩ +≡
box_width: *scaled*; { width of box constructed by *hbox* }
box_height: *scaled*; { height of box constructed by *hbox* }
box_depth: *scaled*; { depth of box constructed by *hbox* }
lig_stack: **array** [1 .. *lig_lookahead*] **of** *quarterword*; { inserted ligature chars }
dummy_info: *four_quarters*; { fake *char_info* for nonexistent character }
suppress_lig: *boolean*; { should we bypass checking for ligatures next time? }

118. ⟨ Set initial values 13 ⟩ +≡
dummy_info.b0 ← *qi*(0); *dummy_info.b1* ← *qi*(0); *dummy_info.b2* ← *qi*(0); *dummy_info.b3* ← *qi*(0);

119. ⟨ Typeset a space in font f and advance k 119 ⟩ ≡
begin *box_width* ← *box_width* + *space*(*f*);
if *send_it* **then**
begin *dvi_out*(*right4*); *dvi_four*(*space*(*f*));
end;
incr(*k*);
end
This code is used in section 116.

120. ⟨ If there's a ligature or kern at the cursor position, update the cursor data structures, possibly advancing k; continue until the cursor wants to move right 120 ⟩ ≡
if (*cur_l* < *font_bc*[*f*]) ∨ (*cur_l* > *font_ec*[*f*]) **then**
begin *i* ← *dummy_info*;
if *cur_l* = 256 **then** *l* ← *bchar_label*[*f*] **else** *l* ← *non_address*;
end
else begin *i* ← *char_info*(*f*)(*cur_l*);
if *char_tag*(*i*) ≠ *lig_tag* **then** *l* ← *non_address*
else begin *l* ← *lig_kern_start*(*f*)(*i*); *j* ← *font_info*[*l*].*qqqq*;
if *skip_byte*(*j*) > *stop_flag* **then** *l* ← *lig_kern_restart*(*f*)(*j*);
end;
end;
if *suppress_lig* **then** *suppress_lig* ← *false*
else while *l* < *qi*(*kern_base*[*f*]) **do**
begin *j* ← *font_info*[*l*].*qqqq*;
if *next_char*(*j*) = *cur_r* **then**
if *skip_byte*(*j*) ≤ *stop_flag* **then**
if *op_byte*(*j*) ≥ *kern_flag* **then**
begin *kern_amount* ← *char_kern*(*f*)(*j*); **goto** *done*;
end
else ⟨ Carry out a ligature operation, updating the cursor structure and possibly advancing k; **goto** *continue* if the cursor doesn't advance, otherwise **goto** *done* 122 ⟩;
if *skip_byte*(*j*) ≥ *stop_flag* **then goto** *done*;
l ← *l* + *skip_byte*(*j*) + 1;
end;
done:
This code is used in section 116.

121. At this point i contains *char_info* for *cur_l*.

```
⟨ Typeset character cur_l, if it exists in the font; also append an optional kern 121 ⟩ ≡
  if char_exists(i) then
    begin box_width ← box_width + char_width(f)(i) + kern_amount;
    hd ← height_depth(i); x ← char_height(f)(hd);
    if x > box_height then box_height ← x;
    x ← char_depth(f)(hd);
    if x > box_depth then box_depth ← x;
    if send_it then
      begin typeset(cur_l);
      if kern_amount ≠ 0 then
        begin dvi_out(right4); dvi_four(kern_amount);
        end;
      end;
    kern_amount ← 0;
    end
```

This code is used in section 116.

122.

```
define pop_stack ≡
    begin decr(stack_ptr);
    if stack_ptr > 0 then cur_r ← lig_stack[stack_ptr]
    else set_cur_r;
    end
```

```
⟨ Carry out a ligature operation, updating the cursor structure and possibly advancing k; goto continue if
      the cursor doesn't advance, otherwise goto done 122 ⟩ ≡
  begin case op_byte(j) of
  1, 5: cur_l ← qo(rem_byte(j));
  2, 6: begin cur_r ← rem_byte(j);
    if stack_ptr = 0 then
      begin stack_ptr ← 1;
      if k < end_k then incr(k)   { a non-space character is consumed }
      else bchar ← non_char;   { the right boundary character is consumed }
      end;
    lig_stack[stack_ptr] ← cur_r;
    end;
  3, 7, 11: begin cur_r ← rem_byte(j); incr(stack_ptr); lig_stack[stack_ptr] ← cur_r;
    if op_byte(j) = 11 then suppress_lig ← true;
    end;
  othercases begin cur_l ← qo(rem_byte(j));
    if stack_ptr > 0 then pop_stack
    else if k = end_k then goto done
      else begin incr(k); set_cur_r;
        end;
    end
  endcases;
  if op_byte(j) > 3 then goto done;
  goto continue;
  end
```

This code is used in section 120.

123. ⟨ Move the cursor to the right and **goto** *continue*, if there's more work to do in the current word 123 ⟩ ≡

```
cur_l ← qo(cur_r);
if stack_ptr > 0 then
  begin pop_stack; goto continue;
  end;
if k < end_k then
  begin incr(k); set_cur_r; goto continue;
  end
```

This code is used in section 116.

124. Gray fonts. A proof diagram constructed by `GFtoDVI` can be regarded as an array of rectangles, where each rectangle is either blank or filled with a special symbol that we shall call x. A blank rectangle represents a white pixel, while x represents a black pixel. Additional labels and reference lines are often superimposed on this array of rectangles; hence it is usually best to choose a symbol x that has a somewhat gray appearance, although any symbol can actually be used.

In order to construct such proofs, `GFtoDVI` needs to work with a special type of font known as a "gray font"; it's possible to obtain a wide variety of different sorts of proofs by using different sorts of gray fonts. The next few paragraphs explain exactly what gray fonts are supposed to contain, in case you want to design your own.

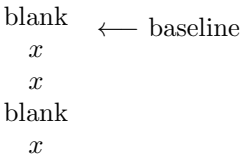
125. The simplest gray font contains only two characters, namely x and another symbol that is used for dots that identify key points. If proofs with relatively large pixels are desired, a two-character gray font is all that's needed. However, if the pixel size is to be relatively small, practical considerations make a two-character font too inefficient, since it requires the typesetting of tens of thousands of tiny little characters; printing device drivers rarely work very well when they are presented with data that is so different from ordinary text. Therefore a gray font with small pixels usually has a number of characters that replicate x in such a way that comparatively few characters actually need to be typeset.

Since many printing devices are not able to cope with arbitrarily large or complex characters, it is not possible for a single gray font to work well on all machines. In fact, x must have a width that is an integer multiple of the printing device's unit of horizontal position, since rounding the positions of grey characters would otherwise produce unsightly streaks on proof output. Thus, there is no way to make the gray font as device-independent as the rest of the system, in the sense that we would expect approximately identical output on machines with different resolution. Fortunately, proof sheets are rarely considered to be final documents; hence `GFtoDVI` is set up to provide results that adapt suitably to local conditions.

126. With such constraints understood, we can now take a look at what `GFtoDVI` expects to see in a gray font. The character x always appears in position 1. It must have positive height h and positive width w; its depth and italic correction are ignored.

Positions 2–120 of a gray font are reserved for special combinations of x's and blanks, stacked on top of each other. None of these character codes need be present in the font; but if they are, the slots should be occupied by characters of width w that have certain configurations of x's and blanks, prescribed for each character position. For example, position 3 of the font should either contain no character at all, or it should contain a character consisting of two x's, one above the other; one of these x's should appear immediately above the baseline, and the other should appear immediately below.

It will be convenient to use a horizontal notation like '`XOXXO`' to stand for a vertical stack of x's and blanks. The convention will be that the stack is built from bottom to top, and the topmost rectangle should sit on the baseline. Thus, '`XOXXO`' stands actually for a character of depth $4h$ that looks like this:



(We use a horizontal notation instead of a vertical one in this explanation, because column vectors take too much space, and because the horizontal notation corresponds to binary numbers in a convenient way.)

Positions 1–63 of a gray font are reserved for the patterns `X`, `XO`, `XX`, `XOO`, `XOX`, ..., `XXXXXX`, just as in the normal binary notation of the numbers 1–63. Positions 64–70 are reserved for the special patterns `XOOOOOO`, `XXOOOOO`, ..., `XXXXXXO`, `XXXXXXX` of length seven; positions 71–78 are, similarly, reserved for the length-eight patterns `XOOOOOOO` through `XXXXXXXX`. The length-nine patterns `XOOOOOOOO` through `XXXXXXXXX` are assigned to positions 79–87, the length-ten patterns to positions 88–97, the length-eleven patterns to positions 98–108, and the length-twelve patterns to positions 109–120.

The following program sets a global array $c[1 \,..\, 120]$ to the bit patterns just described. Another array $d[1 \,..\, 120]$ is set to contain only the next higher bit; this determines the depth of the corresponding character.

```
⟨ Set initial values 13 ⟩ +≡
  c[1] ← 1; d[1] ← 2; two_to_the[0] ← 1; m ← 1;
  for k ← 1 to 13 do two_to_the[k] ← 2 * two_to_the[k − 1];
  for k ← 2 to 6 do ⟨ Add a full set of k-bit characters 128 ⟩;
  for k ← 7 to 12 do ⟨ Add special k-bit characters of the form X..XO..O 129 ⟩;
```

127.
```
⟨ Globals in the outer block 12 ⟩ +≡
c: array [1 .. 120] of 1 .. 4095;   { bit patterns for a gray font }
d: array [1 .. 120] of 2 .. 4096;   { the superleading bits }
two_to_the: array [0 .. 13] of 1 .. 8192;   { powers of 2 }
```

128.
```
⟨ Add a full set of k-bit characters 128 ⟩ ≡
  begin n ← two_to_the[k − 1];
  for j ← 0 to n − 1 do
    begin incr(m); c[m] ← m; d[m] ← n + n;
    end;
  end
```

This code is used in section 126.

129. ⟨ Add special k-bit characters of the form `X..XO..O` 129 ⟩ ≡

```
begin n ← two_to_the[k − 1];
for j ← k downto 1 do
  begin incr(m); d[m] ← n + n;
  if j = k then c[m] ← n
  else c[m] ← c[m − 1] + two_to_the[j − 1];
  end;
end
```

This code is used in section 126.

130. Position 0 of a gray font is reserved for the "dot" character, which should have positive height h' and positive width w'. When `GFtoDVI` wants to put a dot at some place (x, y) on the figure, it positions the dot character so that its reference point is at (x, y). The dot will be considered to occupy a rectangle $(x + \delta, y + \epsilon)$ for $-w' \le \delta \le w'$ and $-h' \le \epsilon \le h'$; the rectangular box for a label will butt up against the rectangle enclosing the dot.

131. All other character positions of a gray font (namely, positions 121–255) are unreserved, in the sense that they have no predefined meaning. But `GFtoDVI` may access them via the "character list" feature of `TFM` files, starting with any of the characters in positions 1–120. In such a case each succeeding character in a list should be equivalent to two of its predecessors, horizontally adjacent to each other. For example, in a character list like

53, 121, 122, 123

character 121 will stand for two 53's, character 122 for two 121's (i.e., four 53's), and character 123 for two 122's (i.e., eight 53's). Since position 53 contains the pattern `XXOXOX`, character 123 in this example would have height h, depth $5h$, and width $8w$, and it would stand for the pattern

$xxxxxxxx$

$xxxxxxxx$

$xxxxxxxx$
$xxxxxxxx$

Such a pattern is, of course, rather unlikely to occur in a `GF` file, but `GFtoDVI` would be able to use if it were present. Designers of gray fonts should provide characters only for patterns that they think will occur often enough to make the doubling worthwhile. For example, the character in position 120 (`XXXXXXXXXXXX`), or whatever is the tallest stack of x's present in the font, is a natural candidate for repeated doubling.

Here's how `GFtoDVI` decides what characters of the gray font will be used, given a configuration of black and white pixels: If there are no black pixels, stop. Otherwise look at the top row that contains at least one black pixel, and the eleven rows that follow. For each such column, find the largest k such that $1 \le k \le 120$ and the gray font contains character k and the pattern assigned to position k appears in the given column. Typeset character k (unless no such character exists) and erase the corresponding black pixels; use doubled characters, if they are present in the gray font, if two or more consecutive equal characters need to be typeset. Repeat the same process on the remaining configuration, until all the black pixels have been erased.

If all characters in positions 1–120 are present, this process is guaranteed to take care of at least six rows each time; and it usually takes care of twelve, since all patterns that contain at most one "run" of x's are present.

132. Fonts have optional parameters, as described in Appendix F of *The TEXbook*, and some of these are important in gray fonts. The slant parameter s, if nonzero, will cause `GFtoDVI` to skew its output; in this case the character x will presumably be a parallelogram with a corresponding slant, rather than the usual rectangle. METAFONT's coordinate (x, y) will appear in physical position $(xw + yhs, yh)$ on the proofsheets.

Parameter number 8 of a gray font specifies the thickness of rules that go on the proofs. If this parameter is zero, TEX's default rule thickness (0.4 pt) will be used.

The other parameters of a gray font are ignored by `GFtoDVI`, but it is conventional to set the font space parameter to w and the xheight parameter to h.

133. For best results the designer of a gray font should choose h and w so that the user's `DVI`-to-hardcopy software will not make any rounding errors. Furthermore, the dot should be an even number $2m$ of pixels in diameter, and the rule thickness should work out to an even number $2n$ of pixels; then the dots and rules will be centered on the correct positions, in case of integer coordinates. Gray fonts are almost always intended for particular output devices, even though '`DVI`' stands for 'device independent'; we use `DVI` files for METAFONT proofs chiefly because software to print `DVI` files is already in place.

134. Slant fonts. GFtoDVI also makes use of another special type of font, if it is necessary to typeset slanted rules. The format of such so-called "slant fonts" is quite a bit simpler than the format of gray fonts.

A slant font should contain exactly n characters, in positions 1 to n, where the character in position k represents a slanted line k units tall, starting at the baseline. These lines all have a fixed slant ratio s.

The following simple algorithm is used to typeset a rule that is m units high: Compute $q = \lceil m/n \rceil$; then typeset q characters of approximately equal size, namely $(m \bmod q)$ copies of character number $\lceil m/q \rceil$ and $q - (m \bmod q)$ copies of character number $\lfloor m/q \rfloor$. For example, if $n = 15$ and $m = 100$, we have $q = 7$; a 100-unit-high rule will be composed of 7 pieces, using characters 14, 14, 14, 14, 14, 15, 15.

⟨ Globals in the outer block 12 ⟩ +≡
rule_slant: *real*; { the slant ratio s in the slant font, or zero if there is no slant font }
slant_n: *integer*; { the number of characters in the slant font }
slant_unit: *real*; { the number of scaled points in the slant font unit }
slant_reported: *real*; { invalid slant ratio reported to the user }

135. GFtoDVI looks only at the height of character n, so the TFM file need not be accurate about the heights of the other characters. (This is fortunate, since TFM format allows at most 16 different heights per font.)

The width of character k should be k/n times s times the height of character n.

The slant parameter of a slant file should be s. It is customary to set the *default_rule_thickness* parameter (number 8) to the thickness of the slanted rules, but GFtoDVI doesn't look at it.

136. For best results on a particular output device, it is usually wise to choose the 'unit' in the above discussion to be an integer number of pixels, and to make it no larger than the default rule thickness in the gray font being used.

137. ⟨ Initialize global variables that depend on the font data 137 ⟩ ≡
if *length*(*font_name*[*slant_font*]) = 0 **then** *rule_slant* ← 0.0
else begin *rule_slant* ← *slant*(*slant_font*)/*unity*; *slant_n* ← *font_ec*[*slant_font*];
i ← *char_info*(*slant_font*)(*slant_n*); *slant_unit* ← *char_height*(*slant_font*)(*height_depth*(*i*))/*slant_n*;
end;
slant_reported ← 0.0;

See also sections 169, 175, 184, 205, and 206.

This code is used in section 98.

138. The following error message is given when an absent slant has been requested.

procedure *slant_complaint*(*r* : *real*);
begin if *abs*(*r* − *slant_reported*) > 0.001 **then**
begin *print_nl*('Sorry,␣I␣can''t␣make␣diagonal␣rules␣of␣slant␣', *r* : 10 : 5, '!');
slant_reported ← *r*;
end;
end;

139. Representation of rectangles. OK—the preliminary spadework has now been done. We're ready at last to concentrate on `GFtoDVI`'s *raison d'être*.

One of the most interesting tasks remaining is to make a "map" of the labels that have been allocated. There usually aren't a great many labels, so we don't need fancy data structures; but we do make use of linked nodes containing nine fields. The nodes generally represent rectangular boxes according to the following conventions:

xl, *xr*, *yt*, and *yb* are the left, right, top, and bottom locations of a rectangle, expressed in `DVI` coordinates. (This program uses scaled points as `DVI` coordinates. Since `DVI` coordinates increase as one moves down the page, *yb* will be greater than *yt*.)

xx and *yy* are the coordinates of the reference point of a box to be typeset from this node, again in `DVI` coordinates.

prev and *next* point to the predecessor and successor of this node. Sometimes the nodes are singly linked and only *next* is relevant; otherwise the nodes are doubly linked in order of their *yy* coordinates, so that we can move down by going to *next*, or up by going to *prev*.

info is the number of a string associated with this node.

The nine fields of a node appear in nine global arrays. Null pointers are denoted by *null*, which happens to be zero.

define *null* = 0

⟨ Types in the outer block 9 ⟩ +≡
node_pointer = *null* .. *max_labels*;

140. ⟨ Globals in the outer block 12 ⟩ +≡
xl, *xr*, *yt*, *yb*: **array** [1 .. *max_labels*] **of** *scaled*; { boundary coordinates }
xx, *yy*: **array** [0 .. *max_labels*] **of** *scaled*; { reference coordinates }
prev, *next*: **array** [0 .. *max_labels*] **of** *node_pointer*; { links }
info: **array** [1 .. *max_labels*] **of** *str_number*; { associated strings }
max_node: *node_pointer*; { the largest node in use }
max_height: *scaled*; { greatest difference between *yy* and *yt* }
max_depth: *scaled*; { greatest difference between *yb* and *yy* }

141. It's easy to allocate a new node (unless no more room is left):

function *get_avail*: *node_pointer*;
begin *incr*(*max_node*);
if *max_node* = *max_labels* **then** *abort*(´Too␣many␣labels␣and/or␣rules!´);
get_avail ← *max_node*;
end;

142. The doubly linked nodes are sorted by *yy* coordinates so that we don't have to work too hard to find nearest neighbors or to determine if rectangles overlap. The first node in the doubly linked rectangle list is always in location 0, and the last node is always in location *max_labels*; the *yy* coordinates of these nodes are very small and very large, respectively.

define *end_of_list* ≡ *max_labels*

⟨ Set initial values 13 ⟩ +≡
yy[0] ← −´10000000000; *yy*[*end_of_list*] ← ´10000000000;

143. The *node_ins* procedure inserts a new rectangle, represented by node p, into the doubly linked list. There's a second parameter, q; node q should already be in the doubly linked list, preferably with $yy[q]$ near $yy[p]$.

```
procedure node_ins(p, q : node_pointer);
  var r: node_pointer;  { for tree traversal }
  begin if yy[p] ≥ yy[q] then
    begin repeat r ← q; q ← next[q]; until yy[p] ≤ yy[q];
    next[r] ← p; prev[p] ← r; next[p] ← q; prev[q] ← p;
    end
  else begin repeat r ← q; q ← prev[q]; until yy[p] ≥ yy[q];
    prev[r] ← p; next[p] ← r; prev[p] ← q; next[q] ← p;
    end;
  if yy[p] − yt[p] > max_height then max_height ← yy[p] − yt[p];
  if yb[p] − yy[p] > max_depth then max_depth ← yb[p] − yy[p];
  end;
```

144. The data structures need to be initialized for each character in the `GF` file.

```
⟨ Initialize variables for the next character 144 ⟩ ≡
  max_node ← 0; next[0] ← end_of_list; prev[end_of_list] ← 0; max_height ← 0; max_depth ← 0;
```

See also sections 156 and 161.

This code is used in section 219.

145. The *overlap* subroutine determines whether or not the rectangle specified in node p has a nonempty intersection with some rectangle in the doubly linked list. Again q is a parameter that gives us a starting point in the list. We assume that $q \neq end\_of\_list$, so that $next[q]$ is meaningful.

```
function overlap(p, q : node_pointer): boolean;
  label exit;
  var y_thresh: scaled;  { cutoff value to speed the search }
    x_left, x_right, y_top, y_bot: scaled;  { boundaries to test for overlap }
    r: node_pointer;  { runs through the neighbors of q }
  begin x_left ← xl[p]; x_right ← xr[p]; y_top ← yt[p]; y_bot ← yb[p];
  ⟨ Look for overlaps in the successors of node q 146 ⟩;
  ⟨ Look for overlaps in node q and its predecessors 147 ⟩;
  overlap ← false;
exit: end;
```

146.

```
⟨ Look for overlaps in the successors of node q 146 ⟩ ≡
  y_thresh ← y_bot + max_height; r ← next[q];
  while yy[r] < y_thresh do
    begin if y_bot > yt[r] then
      if x_left < xr[r] then
        if x_right > xl[r] then
          if y_top < yb[r] then
            begin overlap ← true; return;
            end;
    r ← next[r];
    end
```

This code is used in section 145.

147. ⟨ Look for overlaps in node q and its predecessors 147 ⟩ ≡

```
y_thresh ← y_top − max_depth; r ← q;
while yy[r] > y_thresh do
  begin if y_bot > yt[r] then
    if x_left < xr[r] then
      if x_right > xl[r] then
        if y_top < yb[r] then
          begin overlap ← true; return;
          end;
  r ← prev[r];
  end
```

This code is used in section 145.

148. Nodes that represent dots instead of labels satisfy the following constraints:

$$\begin{aligned} &info[p] < 0; && p \ge first\_dot;\\ &xl[p] = xx[p] - dot\_width, && xr[p] = xx[p] + dot\_width;\\ &yt[p] = yy[p] - dot\_height, && yb[p] = yy[p] + dot\_height. \end{aligned}$$

The *nearest_dot* subroutine finds a node whose reference point is as close as possible to a given position, ignoring nodes that are too close. More precisely, the "nearest" node minimizes

$$d(q,p) = \max\bigl(|xx[q] - xx[p]|, |yy[q] - yy[p]|\bigr)$$

over all nodes q with $d(q,p) \ge d0$. We call the subroutine *nearest_dot* because it is used only when the doubly linked list contains nothing but dots.

The routine also sets the global variable *twin* to *true*, if there is a node $q \ne p$ with $d(q,p) < d0$.

149. ⟨ Globals in the outer block 12 ⟩ +≡

```
first_dot: node_pointer;  { the node address where dots begin }
twin: boolean;  { is there a nearer dot than the "nearest" dot? }
```

150. If there is no nearest dot, the value *null* is returned; otherwise a pointer to the nearest dot is returned.

```
function nearest_dot(p : node_pointer; d0 : scaled): node_pointer;
  var best_q: node_pointer;  { value to return }
    d_min, d: scaled;  { distances }
  begin twin ← false; best_q ← 0; d_min ← ´2000000000;
  ⟨ Search for the nearest dot in nodes following p 151 ⟩;
  ⟨ Search for the nearest dot in nodes preceding p 152 ⟩;
  nearest_dot ← best_q;
  end;
```

151. ⟨ Search for the nearest dot in nodes following p 151 ⟩ ≡

```
q ← next[p];
while yy[q] < yy[p] + d_min do
  begin d ← abs(xx[q] − xx[p]);
  if d < yy[q] − yy[p] then d ← yy[q] − yy[p];
  if d < d0 then twin ← true
  else if d < d_min then
      begin d_min ← d; best_q ← q;
      end;
  q ← next[q];
  end
```

This code is used in section 150.

152. ⟨ Search for the nearest dot in nodes preceding p 152 ⟩ ≡

```
q ← prev[p];
while yy[q] > yy[p] − d_min do
  begin d ← abs(xx[q] − xx[p]);
  if d < yy[p] − yy[q] then d ← yy[p] − yy[q];
  if d < d0 then twin ← true
  else if d < d_min then
      begin d_min ← d; best_q ← q;
      end;
  q ← prev[q];
  end
```

This code is used in section 150.

153. Doing the labels. Each "character" in the GF file is preceded by a number of special commands that define labels, titles, rules, etc. We store these away, to be considered later when the *boc* command appears. The *boc* command establishes the size information by which labels and rules can be positioned, so we spew out the label information as soon as we see the *boc*. The gray pixels will be typeset after all the labels for a particular character have been finished.

154. Here is the part of GFtoDVI that stores information preceding a *boc*. It comes into play when *cur_gf* is between *xxx1* and *no_op*, inclusive.

```
define font_change(#) ≡
        if fonts_not_loaded then
          begin #;
          end
        else print_nl('(Tardy␣font␣change␣will␣be␣ignored␣(byte␣', cur_loc : 1, ')!)')
⟨ Process a no-op command 154 ⟩ ≡
  begin k ← interpret_xxx;
  case k of
  no_operation: do_nothing;
  title_font, label_font, gray_font, slant_font: font_change(font_name[k] ← cur_string;
        font_area[k] ← null_string; font_at[k] ← 0; init_str_ptr ← str_ptr);
  title_font + area_code, label_font + area_code, gray_font + area_code, slant_font + area_code:
        font_change(font_area[k − area_code] ← cur_string; init_str_ptr ← str_ptr);
  title_font + at_code, label_font + at_code, gray_font + at_code, slant_font + at_code:
        font_change(font_at[k − at_code] ← get_yyy; init_str_ptr ← str_ptr);
  rule_thickness_code: rule_thickness ← get_yyy;
  rule_code: ⟨ Store a rule 159 ⟩;
  offset_code: ⟨ Override the offsets 157 ⟩;
  x_offset_code: x_offset ← get_yyy;
  y_offset_code: y_offset ← get_yyy;
  title_code: ⟨ Store a title 162 ⟩;
  null_string: ⟨ Store a label 163 ⟩;
  end; { there are no other cases }
  end
```

This code is used in section 219.

155. The following quantities are cleared just before reading the GF commands pertaining to a character.

```
⟨ Globals in the outer block 12 ⟩ +≡
rule_thickness: scaled; { the current rule thickness (zero means use the default) }
offset_x, offset_y: scaled; { the current offsets for images }
x_offset, y_offset: scaled; { the current offsets for labels }
pre_min_x, pre_max_x, pre_min_y, pre_max_y: scaled;
        { extreme values of coordinates preceding a character, in METAFONT pixels }
```

156.

```
⟨ Initialize variables for the next character 144 ⟩ +≡
  rule_thickness ← 0; offset_x ← 0; offset_y ← 0; x_offset ← 0; y_offset ← 0; pre_min_x ← ´2000000000;
  pre_max_x ← −´2000000000; pre_min_y ← ´2000000000; pre_max_y ← −´2000000000;
```

157.

```
⟨ Override the offsets 157 ⟩ ≡
  begin offset_x ← get_yyy; offset_y ← get_yyy;
  end
```

This code is used in section 154.

158. Rules that will need to be drawn are kept in a linked list accessible via *rule_ptr*, in last-in-first-out order. The nodes of this list will never get into the doubly linked list, and indeed these nodes use different field conventions entirely (because rules may be slanted).

define *x0* ≡ *xl* { starting x coordinate of a stored rule }
define *y0* ≡ *yt* { starting y coordinate (in scaled METAFONT pixels) }
define *x1* ≡ *xr* { ending x coordinate of a stored rule }
define *y1* ≡ *yb* { ending y coordinate of a stored rule }
define *rule_size* ≡ *xx* { thickness of a stored rule, in scaled points }

⟨ Globals in the outer block 12 ⟩ +≡
rule_ptr: *node_pointer*; { top of the stack of remembered rules }

159. ⟨ Store a rule 159 ⟩ ≡
begin *p* ← *get_avail*; *next*[*p*] ← *rule_ptr*; *rule_ptr* ← *p*;
x0[*p*] ← *get_yyy*; *y0*[*p*] ← *get_yyy*; *x1*[*p*] ← *get_yyy*; *y1*[*p*] ← *get_yyy*;
if *x0*[*p*] < *pre_min_x* **then** *pre_min_x* ← *x0*[*p*];
if *x0*[*p*] > *pre_max_x* **then** *pre_max_x* ← *x0*[*p*];
if *y0*[*p*] < *pre_min_y* **then** *pre_min_y* ← *y0*[*p*];
if *y0*[*p*] > *pre_max_y* **then** *pre_max_y* ← *y0*[*p*];
if *x1*[*p*] < *pre_min_x* **then** *pre_min_x* ← *x1*[*p*];
if *x1*[*p*] > *pre_max_x* **then** *pre_max_x* ← *x1*[*p*];
if *y1*[*p*] < *pre_min_y* **then** *pre_min_y* ← *y1*[*p*];
if *y1*[*p*] > *pre_max_y* **then** *pre_max_y* ← *y1*[*p*];
rule_size[*p*] ← *rule_thickness*;
end

This code is used in section 154.

160. Titles and labels are, likewise, stored temporarily in singly linked lists. In this case the lists are first-in-first-out. Variables *title_tail* and *label_tail* point to the most recently inserted title or label; variables *title_head* and *label_head* point to the beginning of the list. (A standard coding trick is used for *label_head*, which is kept in *next*[*end_of_list*]; we have *label_tail* = *end_of_list* when the list is empty.)

The *prev* field in nodes of the temporary label list specifies the type of label, so we call it *lab_typ*.

define *lab_typ* ≡ *prev* { the type of a stored label (`"/"` . . `"8"`) }
define *label_head* ≡ *next*[*end_of_list*]

⟨ Globals in the outer block 12 ⟩ +≡
label_tail: *node_pointer*; { tail of the queue of remembered labels }
title_head, *title_tail*: *node_pointer*; { head and tail of the queue for titles }

161. We must start the lists out empty.

⟨ Initialize variables for the next character 144 ⟩ +≡
rule_ptr ← *null*; *title_head* ← *null*; *title_tail* ← *null*; *label_head* ← *null*; *label_tail* ← *end_of_list*;
first_dot ← *max_labels*;

162. ⟨ Store a title 162 ⟩ ≡
begin *p* ← *get_avail*; *info*[*p*] ← *cur_string*;
if *title_head* = *null* **then** *title_head* ← *p*
else *next*[*title_tail*] ← *p*;
title_tail ← *p*;
end

This code is used in section 154.

163. We store the coordinates of each label in units of METAFONT pixels; they will be converted to DVI coordinates later.

```
⟨ Store a label 163 ⟩ ≡
  if (label_type < "/") ∨ (label_type > "8") then
    print_nl('Bad␣label␣type␣precedes␣byte␣', cur_loc : 1, '!')
  else begin p ← get_avail; next[label_tail] ← p; label_tail ← p;
    lab_typ[p] ← label_type; info[p] ← cur_string;
    xx[p] ← get_yyy; yy[p] ← get_yyy;
    if xx[p] < pre_min_x then pre_min_x ← xx[p];
    if xx[p] > pre_max_x then pre_max_x ← xx[p];
    if yy[p] < pre_min_y then pre_min_y ← yy[p];
    if yy[p] > pre_max_y then pre_max_y ← yy[p];
    end
```

This code is used in section 154.

164. The process of ferreting everything away comes to an abrupt halt when a *boc* command is sensed. The following steps are performed at such times:

```
⟨ Process a character 164 ⟩ ≡
  begin check_fonts; ⟨ Finish reading the parameters of the boc 165 ⟩;
  ⟨ Get ready to convert METAFONT coordinates to DVI coordinates 170 ⟩;
  ⟨ Output the bop and the title line 172 ⟩;
  print('[', total_pages : 1); update_terminal; { print a progress report }
  ⟨ Output all rules for the current character 173 ⟩;
  ⟨ Output all labels for the current character 181 ⟩;
  do_pixels; dvi_out(eop); { finish the page }
  ⟨ Adjust the maximum page width 203 ⟩;
  print(']'); update_terminal;
  end
```

This code is used in section 219.

165.

```
⟨ Finish reading the parameters of the boc 165 ⟩ ≡
  if cur_gf = boc then
    begin ext ← signed_quad; { read the character code }
    char_code ← ext mod 256;
    if char_code < 0 then char_code ← char_code + 256;
    ext ← (ext − char_code) div 256; k ← signed_quad; { read and ignore the prev pointer }
    min_x ← signed_quad; { read the minimum x coordinate }
    max_x ← signed_quad; { read the maximum x coordinate }
    min_y ← signed_quad; { read the minimum y coordinate }
    max_y ← signed_quad; { read the maximum y coordinate }
    end
  else begin ext ← 0; char_code ← get_byte; { cur_gf = boc1 }
    min_x ← get_byte; max_x ← get_byte; min_x ← max_x − min_x;
    min_y ← get_byte; max_y ← get_byte; min_y ← max_y − min_y;
    end;
  if max_x − min_x > widest_row then abort('Character␣too␣wide!')
```

This code is used in section 164.

166. ⟨ Globals in the outer block 12 ⟩ +≡
char_code, *ext*: *integer*; { the current character code and extension }
min_x, *max_x*, *min_y*, *max_y*: *integer*; { character boundaries, in pixels }
x, *y*: *integer*; { current painting position, in pixels }
z: *integer*; { initial painting position in row, relative to *min_x* }

167. METAFONT coordinates (x, y) are converted to DVI coordinates by the following routine. Real values *x_ratio*, *y_ratio*, and *slant_ratio* will have been calculated based on the gray font; *scaled* values *delta_x* and *delta_y* will have been computed so that, in the absence of slanting and offsets, the METAFONT coordinates $(min\_x, max\_y + 1)$ will correspond to the DVI coordinates $(0, 50\,\text{pt})$.

```
procedure convert(x, y : scaled);
  begin x ← x + x_offset; y ← y + y_offset; dvi_y ← −round(y_ratio ∗ y) + delta_y;
  dvi_x ← round(x_ratio ∗ x + slant_ratio ∗ y) + delta_x;
  end;
```

168. ⟨ Globals in the outer block 12 ⟩ +≡
x_ratio, *y_ratio*, *slant_ratio*: *real*; { conversion factors }
unsc_x_ratio, *unsc_y_ratio*, *unsc_slant_ratio*: *real*; { ditto, times *unity* }
fudge_factor: *real*; { unconversion factor }
delta_x, *delta_y*: *scaled*; { magic constants used by *convert* }
dvi_x, *dvi_y*: *scaled*; { outputs of *convert*, in scaled points }
over_col: *scaled*; { overflow labels start here }
page_height, *page_width*: *scaled*; { size of the current page }

169. ⟨ Initialize global variables that depend on the font data 137 ⟩ +≡

```
i ← char_info(gray_font)(1);
if ¬char_exists(i) then abort('Missing␣pixel␣char!');
unsc_x_ratio ← char_width(gray_font)(i); x_ratio ← unsc_x_ratio/unity;
unsc_y_ratio ← char_height(gray_font)(height_depth(i)); y_ratio ← unsc_y_ratio/unity;
unsc_slant_ratio ← slant(gray_font) ∗ y_ratio; slant_ratio ← unsc_slant_ratio/unity;
if x_ratio ∗ y_ratio = 0 then abort('Vanishing␣pixel␣size!');
fudge_factor ← (slant_ratio/x_ratio)/y_ratio;
```

170. ⟨ Get ready to convert METAFONT coordinates to DVI coordinates 170 ⟩ ≡

```
if pre_min_x < min_x ∗ unity then offset_x ← offset_x + min_x ∗ unity − pre_min_x;
if pre_max_y > max_y ∗ unity then offset_y ← offset_y + max_y ∗ unity − pre_max_y;
if pre_max_x > max_x ∗ unity then pre_max_x ← pre_max_x div unity
else pre_max_x ← max_x;
if pre_min_y < min_y ∗ unity then pre_min_y ← pre_min_y div unity
else pre_min_y ← min_y;
delta_y ← round(unsc_y_ratio ∗ (max_y + 1) − y_ratio ∗ offset_y) + 3276800;
delta_x ← round(x_ratio ∗ offset_x − unsc_x_ratio ∗ min_x);
if slant_ratio ≥ 0 then over_col ← round(unsc_x_ratio ∗ pre_max_x + unsc_slant_ratio ∗ max_y)
else over_col ← round(unsc_x_ratio ∗ pre_max_x + unsc_slant_ratio ∗ min_y);
over_col ← over_col + delta_x + 10000000;
page_height ← round(unsc_y_ratio ∗ (max_y + 1 − pre_min_y)) + 3276800 − offset_y;
if page_height > max_v then max_v ← page_height;
page_width ← over_col − 10000000
```

This code is used in section 164.

171. The *dvi_goto* subroutine outputs bytes to the DVI file that will initiate typesetting at given DVI coordinates, assuming that the current position of the DVI reader is $(0,0)$. This subroutine begins by outputting a *push* command; therefore, a *pop* command should be given later. That *pop* will restore the DVI position to $(0,0)$.

```
procedure dvi_goto(x, y : scaled);
  begin dvi_out(push);
  if x ≠ 0 then
    begin dvi_out(right4); dvi_four(x);
    end;
  if y ≠ 0 then
    begin dvi_out(down4); dvi_four(y);
    end;
  end;
```

172. ⟨ Output the *bop* and the title line 172 ⟩ ≡

```
dvi_out(bop); incr(total_pages); dvi_four(total_pages); dvi_four(char_code); dvi_four(ext);
for k ← 3 to 9 do dvi_four(0);
dvi_four(last_bop); last_bop ← dvi_offset + dvi_ptr − 45;
dvi_goto(0, 655360); { the top baseline is 10 pt down }
if use_logo then
  begin select_font(logo_font); hbox(small_logo, logo_font, true);
  end;
select_font(title_font); hbox(time_stamp, title_font, true);
hbox(page_header, title_font, true); dvi_scaled(total_pages * 65536.0);
if (char_code ≠ 0) ∨ (ext ≠ 0) then
  begin hbox(char_header, title_font, true); dvi_scaled(char_code * 65536.0);
  if ext ≠ 0 then
    begin hbox(ext_header, title_font, true); dvi_scaled(ext * 65536.0);
    end;
  end;
if title_head ≠ null then
  begin next[title_tail] ← null;
  repeat hbox(left_quotes, title_font, true); hbox(info[title_head], title_font, true);
    hbox(right_quotes, title_font, true); title_head ← next[title_head];
  until title_head = null;
  end;
dvi_out(pop)
```

This code is used in section 164.

173. **define** *tol* ≡ 6554 { one tenth of a point, in DVI coordinates }

```
⟨ Output all rules for the current character 173 ⟩ ≡
  if rule_slant ≠ 0 then select_font(slant_font);
  while rule_ptr ≠ null do
    begin p ← rule_ptr; rule_ptr ← next[p];
    if rule_size[p] = 0 then rule_size[p] ← gray_rule_thickness;
    if rule_size[p] > 0 then
      begin convert(x0[p], y0[p]); temp_x ← dvi_x; temp_y ← dvi_y; convert(x1[p], y1[p]);
      if abs(temp_x − dvi_x) < tol then ⟨ Output a vertical rule 176 ⟩
      else if abs(temp_y − dvi_y) < tol then ⟨ Output a horizontal rule 177 ⟩
        else ⟨ Try to output a diagonal rule 178 ⟩;
      end;
    end
```

This code is used in section 164.

174.
```
⟨ Globals in the outer block 12 ⟩ +≡
gray_rule_thickness: scaled; { thickness of rules, according to the gray font }
temp_x, temp_y: scaled; { temporary registers for intermediate calculations }
```

175.
```
⟨ Initialize global variables that depend on the font data 137 ⟩ +≡
  gray_rule_thickness ← default_rule_thickness(gray_font);
  if gray_rule_thickness = 0 then gray_rule_thickness ← 26214; { 0.4 pt }
```

176.
```
⟨ Output a vertical rule 176 ⟩ ≡
  begin if temp_y > dvi_y then
    begin k ← temp_y; temp_y ← dvi_y; dvi_y ← k;
    end;
  dvi_goto(dvi_x − (rule_size[p] div 2), dvi_y); dvi_out(put_rule); dvi_four(dvi_y − temp_y);
  dvi_four(rule_size[p]); dvi_out(pop);
  end
```

This code is used in section 173.

177.
```
⟨ Output a horizontal rule 177 ⟩ ≡
  begin if temp_x < dvi_x then
    begin k ← temp_x; temp_x ← dvi_x; dvi_x ← k;
    end;
  dvi_goto(dvi_x, dvi_y + (rule_size[p] div 2)); dvi_out(put_rule); dvi_four(rule_size[p]);
  dvi_four(temp_x − dvi_x); dvi_out(pop);
  end
```

This code is used in section 173.

178. ⟨ Try to output a diagonal rule 178 ⟩ ≡

```
if (rule_slant = 0) ∨ (abs(temp_x + rule_slant * (temp_y − dvi_y) − dvi_x) > rule_size[p]) then
  slant_complaint((dvi_x − temp_x)/(temp_y − dvi_y))
else begin if temp_y > dvi_y then
    begin k ← temp_y; temp_y ← dvi_y; dvi_y ← k;
    k ← temp_x; temp_x ← dvi_x; dvi_x ← k;
    end;
  m ← round((dvi_y − temp_y)/slant_unit);
  if m > 0 then
    begin dvi_goto(dvi_x, dvi_y); q ← ((m − 1) div slant_n) + 1; k ← m div q; p ← m mod q;
    q ← q − p; ⟨ Vertically typeset q copies of character k 179 ⟩;
    ⟨ Vertically typeset p copies of character k + 1 180 ⟩;
    dvi_out(pop);
    end;
  end
```

This code is used in section 173.

179. ⟨ Vertically typeset q copies of character k 179 ⟩ ≡

```
typeset(k); dy ← round(k * slant_unit); dvi_out(z4); dvi_four(−dy);
while q > 1 do
  begin typeset(k); dvi_out(z0); decr(q);
  end
```

This code is used in section 178.

180. ⟨ Vertically typeset p copies of character $k + 1$ 180 ⟩ ≡

```
if p > 0 then
  begin incr(k); typeset(k); dy ← round(k * slant_unit); dvi_out(z4); dvi_four(−dy);
  while p > 1 do
    begin typeset(k); dvi_out(z0); decr(p);
    end;
  end
```

This code is used in section 178.

181. Now we come to a more interesting part of the computation, where we go through the stored labels and try to fit them in the illustration for the current character, together with their associated dots.

It would simplify font-switching slightly if we were to typeset the labels first, but we find it desirable to typeset the dots first and then turn to the labels. This procedure makes it possible for us to allow the dots to overlap each other without allowing the labels to overlap. After the dots are in place, we typeset all prescribed labels, that is, labels with a *lab_typ* of `"1" .. "8"`; these, too, are allowed to overlap the dots and each other.

⟨ Output all labels for the current character 181 ⟩ ≡

```
overflow_line ← 1;
if label_head ≠ null then
  begin next[label_tail] ← null; select_font(gray_font); ⟨ Output all dots 187 ⟩;
  ⟨ Find nearest dots, to help in label positioning 191 ⟩;
  select_font(label_font); ⟨ Output all prescribed labels 189 ⟩;
  ⟨ Output all attachable labels 193 ⟩;
  ⟨ Output all overflow labels 200 ⟩;
  end
```

This code is used in section 164.

182. ⟨ Globals in the outer block 12 ⟩ +≡
overflow_line: *integer*; { the number of labels that didn't fit, plus 1 }

183. A label that appears above its dot is considered to occupy a rectangle of height $h+\Delta$, depth d, and width $w+2\Delta$, where (h,w,d) are the height, width, and depth of the label computed by *hbox*, and Δ is an additional amount of blank space that keeps labels from coming too close to each other. (`GFtoDVI` arbitrarily defines Δ to be one half the width of a space in the label font.) This label is centered over its dot, with its baseline $d+h'$ above the center of the dot; here $h'=$ *dot_height* is the height of character 0 in the gray font.

Similarly, a label that appears below its dot is considered to occupy a rectangle of height h, depth $d+\Delta$, and width $w+2\Delta$; the baseline is $h+h'$ below the center of the dot.

A label at the right of its dot is considered to occupy a rectangle of height $h+\Delta$, depth $d+\Delta$, and width $w+\Delta$. Its reference point can be found by starting at the center of the dot and moving right $w'=$ *dot_width* (i.e., the width of character 0 in the gray font), then moving down by half the x-height of the label font. A label at the left of its dot is similar.

A dot is considered to occupy a rectangle of height $2h'$ and width $2w'$, centered on the dot.

When the label type is `"1"` or more, the labels are put into the doubly linked list unconditionally. Otherwise they are put into the list only if we can find a way to fit them in without overlapping any previously inserted rectangles.

⟨ Globals in the outer block 12 ⟩ +≡
delta: *scaled*; { extra padding to keep labels from being too close }
half_x_height: *scaled*; { amount to drop baseline of label below the dot center }
thrice_x_height: *scaled*; { baseline separation for overflow labels }
dot_width, *dot_height*: *scaled*; { w' and h' in the discussion above }

184. ⟨ Initialize global variables that depend on the font data 137 ⟩ +≡
$i \leftarrow$ *char_info*(*gray_font*)(0);
if ¬*char_exists*(*i*) **then** *abort*(´`Missing␣dot␣char!`´);
dot_width ← *char_width*(*gray_font*)(*i*); *dot_height* ← *char_height*(*gray_font*)(*height_depth*(*i*));
delta ← *space*(*label_font*) **div** 2; *thrice_x_height* ← 3 ∗ *x_height*(*label_font*);
half_x_height ← *thrice_x_height* **div** 6;

185. Here is a subroutine that computes the rectangle boundaries $xl[p]$, $xr[p]$, $yt[p]$, $yb[p]$, and the reference point coordinates $xx[p]$, $yy[p]$, for a label that is to be placed above a dot. The coordinates of the dot's center are assumed given in *dvi_x* and *dvi_y*; the *hbox* subroutine is assumed to have already computed the height, width, and depth of the label box.

procedure *top_coords*(*p* : *node_pointer*);
begin $xx[p] \leftarrow$ *dvi_x* − (*box_width* **div** 2); $xl[p] \leftarrow xx[p]$ − *delta*; $xr[p] \leftarrow xx[p]$ + *box_width* + *delta*;
$yb[p] \leftarrow$ *dvi_y* − *dot_height*; $yy[p] \leftarrow yb[p]$ − *box_depth*; $yt[p] \leftarrow yy[p]$ − *box_height* − *delta*;
end;

186. The other three label positions are handled by similar routines.

```
procedure bot_coords(p : node_pointer);
  begin xx[p] ← dvi_x − (box_width div 2); xl[p] ← xx[p] − delta; xr[p] ← xx[p] + box_width + delta;
  yt[p] ← dvi_y + dot_height; yy[p] ← yt[p] + box_height; yb[p] ← yy[p] + box_depth + delta;
  end;
procedure right_coords(p : node_pointer);
  begin xl[p] ← dvi_x + dot_width; xx[p] ← xl[p]; xr[p] ← xx[p] + box_width + delta;
  yy[p] ← dvi_y + half_x_height; yb[p] ← yy[p] + box_depth + delta; yt[p] ← yy[p] − box_height − delta;
  end;
procedure left_coords(p : node_pointer);
  begin xr[p] ← dvi_x − dot_width; xx[p] ← xr[p] − box_width; xl[p] ← xx[p] − delta;
  yy[p] ← dvi_y + half_x_height; yb[p] ← yy[p] + box_depth + delta; yt[p] ← yy[p] − box_height − delta;
  end;
```

187.
```
⟨ Output all dots 187 ⟩ ≡
  p ← label_head; first_dot ← max_node + 1;
  while p ≠ null do
    begin convert(xx[p], yy[p]); xx[p] ← dvi_x; yy[p] ← dvi_y;
    if lab_typ[p] < "5" then ⟨ Enter a dot for label p in the rectangle list, and typeset the dot 188 ⟩;
    p ← next[p];
    end
```

This code is used in section 181.

188. We plant links between dots and their labels by using (or abusing) the *xl* and *info* fields, which aren't needed for their normal purposes.

```
  define dot_for_label ≡ xl
  define label_for_dot ≡ info
⟨ Enter a dot for label p in the rectangle list, and typeset the dot 188 ⟩ ≡
  begin q ← get_avail; dot_for_label[p] ← q; label_for_dot[q] ← p;
  xx[q] ← dvi_x; xl[q] ← dvi_x − dot_width; xr[q] ← dvi_x + dot_width;
  yy[q] ← dvi_y; yt[q] ← dvi_y − dot_height; yb[q] ← dvi_y + dot_height;
  node_ins(q, 0);
  dvi_goto(xx[q], yy[q]); dvi_out(0); dvi_out(pop);
  end
```

This code is used in section 187.

189. Prescribed labels are now taken out of the singly linked list and inserted into the doubly linked list.

```
⟨ Output all prescribed labels 189 ⟩ ≡
  q ← end_of_list; { label_head = next[q] }
  while next[q] ≠ null do
    begin p ← next[q];
    if lab_typ[p] > "0" then
      begin next[q] ← next[p];
      ⟨ Enter a prescribed label for node p into the rectangle list, and typeset it 190 ⟩;
      end
    else q ← next[q];
    end
```

This code is used in section 181.

190. ⟨ Enter a prescribed label for node p into the rectangle list, and typeset it 190 ⟩ ≡

```
begin hbox(info[p], label_font, false);  { Compute the size of this label }
dvi_x ← xx[p]; dvi_y ← yy[p];
if lab_typ[p] < "5" then r ← dot_for_label[p] else r ← 0;
case lab_typ[p] of
"1","5": top_coords(p);
"2","6": left_coords(p);
"3","7": right_coords(p);
"4","8": bot_coords(p);
end;  { no other cases are possible }
node_ins(p, r);
dvi_goto(xx[p], yy[p]); hbox(info[p], label_font, true); dvi_out(pop);
end
```

This code is used in section 189.

191. GFtoDVI's algorithm for positioning the "floating" labels was devised by Arthur L. Samuel. It tries to place labels in a priority order, based on the position of the nearest dot to a given dot. If that dot, for example, lies in the first octant (i.e., east to northeast of the given dot), the given label will be put into the west slot unless that slot is already blocked; then the south slot will be tried, etc.

First we need to compute the octants. We also note if two or more dots are nearly coincident, since Samuel's algorithm modifies the priority order on that case. The information is temporarily recorded in the xr array.

```
define octant ≡ xr   { octant code for nearest dot, plus 8 for coincident dots }
```

⟨ Find nearest dots, to help in label positioning 191 ⟩ ≡

```
p ← label_head;
while p ≠ null do
  begin if lab_typ[p] ≤ "0" then ⟨ Compute the octant code for floating label p 192 ⟩;
  p ← next[p];
  end;
```

This code is used in section 181.

192. There's a sneaky way to identify octant numbers, represented by the code shown here. (Remember that y coordinates increase downward in the DVI convention.)

```
define first_octant = 0
define second_octant = 1
define third_octant = 2
define fourth_octant = 3
define fifth_octant = 7
define sixth_octant = 6
define seventh_octant = 5
define eighth_octant = 4
```

⟨ Compute the octant code for floating label p 192 ⟩ ≡

```
begin r ← dot_for_label[p]; q ← nearest_dot(r, 10);
if twin then octant[p] ← 8 else octant[p] ← 0;
if q ≠ null then
  begin dx ← xx[q] − xx[r]; dy ← yy[q] − yy[r];
  if dy > 0 then octant[p] ← octant[p] + 4;
  if dx < 0 then incr(octant[p]);
  if dy > dx then incr(octant[p]);
  if −dy > dx then incr(octant[p]);
  end;
end
```

This code is used in section 191.

193. A procedure called *place_label* will try to place the remaining labels in turn. If it fails, we "disconnect" the dot from this label so that an unlabeled dot will not appear as a reference in the overflow column.

⟨ Output all attachable labels 193 ⟩ ≡

```
q ← end_of_list;  { now next[q] = label_head }
while next[q] ≠ null do
  begin p ← next[q]; r ← next[p]; s ← dot_for_label[p];
  if place_label(p) then next[q] ← r
  else begin label_for_dot[s] ← null;  { disconnect the dot }
    if lab_typ[p] = "/" then next[q] ← r  { remove label from list }
    else q ← p;  { retain label in list for the overflow column }
    end;
  end
```

This code is used in section 181.

194. Here is the *place_label* routine, which uses the previously computed *octant* information as a heuristic. If the label can be placed, it is inserted into the rectangle list and typeset.

```
function place_label(p : node_pointer): boolean;
  label exit, found;
  var oct: 0 .. 15;  { octant code }
    dfl: node_pointer;  { saved value of dot_for_label[p] }
  begin hbox(info[p], label_font, false);  { Compute the size of this label }
  dvi_x ← xx[p]; dvi_y ← yy[p]; ⟨ Find non-overlapping coordinates, if possible, and goto found; otherwise
      set place_label ← false and return 195 ⟩;
found: node_ins(p, dfl);
  dvi_goto(xx[p], yy[p]); hbox(info[p], label_font, true); dvi_out(pop); place_label ← true;
exit: end;
```

195. ⟨ Find non-overlapping coordinates, if possible, and **goto** found; otherwise set *place_label* ← *false* and **return** 195 ⟩ ≡

```
dfl ← dot_for_label[p]; oct ← octant[p]; ⟨ Try the first choice for label direction 196 ⟩;
⟨ Try the second choice for label direction 197 ⟩;
⟨ Try the third choice for label direction 198 ⟩;
⟨ Try the fourth choice for label direction 199 ⟩;
xx[p] ← dvi_x; yy[p] ← dvi_y; dot_for_label[p] ← dfl;   { no luck; restore the coordinates }
place_label ← false; return
```

This code is used in section 194.

196. ⟨ Try the first choice for label direction 196 ⟩ ≡

```
case oct of
first_octant, eighth_octant, second_octant + 8, seventh_octant + 8: left_coords(p);
second_octant, third_octant, first_octant + 8, fourth_octant + 8: bot_coords(p);
fourth_octant, fifth_octant, third_octant + 8, sixth_octant + 8: right_coords(p);
sixth_octant, seventh_octant, fifth_octant + 8, eighth_octant + 8: top_coords(p);
end;
if ¬overlap(p, dfl) then goto found
```

This code is used in section 195.

197. ⟨ Try the second choice for label direction 197 ⟩ ≡

```
case oct of
first_octant, fourth_octant, fifth_octant + 8, eighth_octant + 8: bot_coords(p);
second_octant, seventh_octant, third_octant + 8, sixth_octant + 8: left_coords(p);
third_octant, sixth_octant, second_octant + 8, seventh_octant + 8: right_coords(p);
fifth_octant, eighth_octant, first_octant + 8, fourth_octant + 8: top_coords(p);
end;
if ¬overlap(p, dfl) then goto found
```

This code is used in section 195.

198. ⟨ Try the third choice for label direction 198 ⟩ ≡

```
case oct of
first_octant, fourth_octant, sixth_octant + 8, seventh_octant + 8: top_coords(p);
second_octant, seventh_octant, fourth_octant + 8, fifth_octant + 8: right_coords(p);
third_octant, sixth_octant, first_octant + 8, eighth_octant + 8: left_coords(p);
fifth_octant, eighth_octant, second_octant + 8, third_octant + 8: bot_coords(p);
end;
if ¬overlap(p, dfl) then goto found
```

This code is used in section 195.

199. ⟨ Try the fourth choice for label direction 199 ⟩ ≡

```
case oct of
first_octant, eighth_octant, first_octant + 8, eighth_octant + 8: right_coords(p);
second_octant, third_octant, second_octant + 8, third_octant + 8: top_coords(p);
fourth_octant, fifth_octant, fourth_octant + 8, fifth_octant + 8: left_coords(p);
sixth_octant, seventh_octant, sixth_octant + 8, seventh_octant + 8: bot_coords(p);
end;
if ¬overlap(p, dfl) then goto found
```

This code is used in section 195.

200. ⟨ Output all overflow labels 200 ⟩ ≡
```
⟨ Remove all rectangles from list, except for dots that have labels 201 ⟩;
p ← label_head;
while p ≠ null do
  begin ⟨ Typeset an overflow label for p 202 ⟩;
  p ← next[p];
  end
```
This code is used in section 181.

201. When we remove a dot that couldn't be labeled, we set its *next* field to the preceding node that survives, so that we can use the *nearest_dot* routine later. (This is a bit of a kludge.)

⟨ Remove all rectangles from list, except for dots that have labels 201 ⟩ ≡
```
p ← next[0];
while p ≠ end_of_list do
  begin q ← next[p];
  if (p < first_dot) ∨ (label_for_dot[p] = null) then
    begin r ← prev[p]; next[r] ← q; prev[q] ← r; next[p] ← r;
    end;
  p ← q;
  end
```
This code is used in section 200.

202. Now we have to insert p into the list temporarily, because of the way *nearest_dot* works.

⟨ Typeset an overflow label for p 202 ⟩ ≡
```
begin r ← next[dot_for_label[p]]; s ← next[r]; t ← next[p]; next[p] ← s; prev[s] ← p; next[r] ← p;
prev[p] ← r;
q ← nearest_dot(p, 0);
next[r] ← s; prev[s] ← r; next[p] ← t;   { remove p again }
incr(overflow_line); dvi_goto(over_col, overflow_line * thrice_x_height + 655360);
hbox(info[p], label_font, true);
if q ≠ null then
  begin hbox(equals_sign, label_font, true); hbox(info[label_for_dot[q]], label_font, true);
  hbox(plus_sign, label_font, true); dvi_scaled((xx[p] − xx[q])/x_ratio + (yy[p] − yy[q]) * fudge_factor);
  dvi_out("⌴,"); dvi_scaled((yy[q] − yy[p])/y_ratio); dvi_out("⌴)");
  end;
dvi_out(pop);
end
```
This code is used in section 200.

203. ⟨ Adjust the maximum page width 203 ⟩ ≡
```
if overflow_line > 1 then page_width ← over_col + 10000000;
      { overflow labels are estimated to occupy 10^7 sp }
if page_width > max_h then max_h ← page_width
```
This code is used in section 164.

204. Doing the pixels. The most interesting part of `GFtoDVI` is the way it makes use of a gray font to typeset the pixels of a character. In fact, the author must admit having great fun devising the algorithms below. Perhaps the reader will also enjoy reading them.

The basic idea will be to use an array of 12-bit integers to represent the next twelve rows that need to be typeset. The binary expansions of these integers, reading from least significant bit to most significant bit, will represent pixels from top to bottom.

205. We have already used such a binary representation in the tables $c[1\,..\,120]$ and $d[1\,..\,120]$ of bit patterns and lengths that are potentially present in a gray font; we shall now use those tables to compute an auxiliary array $b[0\,..\,4095]$. Given a 12-bit number v, the gray-font character appropriate to v's binary pattern will be $b[v]$. If no character should be typeset for this pattern in the current row, $b[v]$ will be 0.

The array b can have many different configurations, depending on how many characters are actually present in the gray font. But it's not difficult to compute b by going through the existing characters in increasing order and marking all patterns x to which they apply.

```
⟨ Initialize global variables that depend on the font data 137 ⟩ +≡
  for k ← 0 to 4095 do b[k] ← 0;
  for k ← font_bc[gray_font] to font_ec[gray_font] do
    if k ≥ 1 then
      if k ≤ 120 then
        if char_exists(char_info(gray_font)(k)) then
          begin v ← c[k];
          repeat b[v] ← k; v ← v + d[k];
          until v > 4095;
          end;
```

206. We also compute an auxiliary array $rho[0\,..\,4095]$ such that $rho[v] = 2^j$ when v is an odd multiple of 2^j; we also set $rho[0] = 2^{12}$.

```
⟨ Initialize global variables that depend on the font data 137 ⟩ +≡
  for j ← 0 to 11 do
    begin k ← two_to_the[j]; v ← k;
    repeat rho[v] ← k; v ← v + k + k;
    until v > 4095;
    end;
  rho[0] ← 4096;
```

207.
```
⟨ Globals in the outer block 12 ⟩ +≡
b: array [0 .. 4095] of 0 .. 120;   { largest existing character for a given pattern }
rho: array [0 .. 4095] of 1 .. 4096;   { the "ruler function" }
```

208. But how will we use these tables? Let's imagine that the DVI file already contains instructions that have selected the gray font and moved to the proper horizontal coordinate for the row that we wish to process next. Let's suppose that 12-bit patterns have been set up in array a, and that the global variables *starting_col* and *finishing_col* are known such that $a[j]$ is zero unless $starting\_col \le j \le finishing\_col$. Here's what we can do, assuming that appropriate local variables and labels have been declared:

```
⟨ Typeset the pixels of the current row 208 ⟩ ≡
  j ← starting_col;
  loop begin while (j ≤ finishing_col) ∧ (b[a[j]] = 0) do incr(j);
    if j > finishing_col then goto done;
    dvi_out(push); ⟨ Move to column j in the DVI output 209 ⟩;
    repeat v ← b[a[j]]; a[j] ← a[j] − c[v]; k ← j; incr(j);
      while b[a[j]] = v do
        begin a[j] ← a[j] − c[v]; incr(j);
        end;
      k ← j − k; ⟨ Output the equivalent of k copies of character v 210 ⟩;
    until b[a[j]] = 0;
    dvi_out(pop);
    end;
done:
```

This code is used in section 218.

209.

```
⟨ Move to column j in the DVI output 209 ⟩ ≡
  dvi_out(right4); dvi_four(round(unsc_x_ratio * j + unsc_slant_ratio * y) + delta_x)
```

This code is used in section 208.

210. The doubling-up property of gray font character lists is utilized here.

```
⟨ Output the equivalent of k copies of character v 210 ⟩ ≡
reswitch: if k = 1 then typeset(v)
  else begin i ← char_info(gray_font)(v);
    if char_tag(i) = list_tag then { v has a successor }
      begin if odd(k) then typeset(v);
      k ← k div 2; v ← qo(rem_byte(i)); goto reswitch;
      end
    else repeat typeset(v); decr(k);
      until k = 0;
    end
```

This code is used in section 208.

211.

```
⟨ Globals in the outer block 12 ⟩ +≡
a: array [0 .. widest_row] of 0 .. 4095; { bit patterns for twelve rows }
```

212. In order to use the approach above, we need to be able to initialize array a, and we need to be able to keep it up to date as new rows scroll by. A moment's thought about the problem reveals that we will either have to read an entire character from the GF file into memory, or we'll need to adopt a coroutine-like approach: A single *skip* command in the GF file might need to be processed in pieces, since it might generate more rows of zeroes than we are ready to absorb all at once into a.

The coroutine method actually turns out to be quite simple, so we shall introduce a global variable *blank_rows*, which tells how many rows of blanks should be generated before we read the GF instructions for another row.

```
⟨ Globals in the outer block 12 ⟩ +≡
blank_rows: integer; { rows of blanks carried over from a previous GF command }
```

213. Initialization and updating of a can now be handled as follows, if we introduce another variable l that is set initially to 1:

```
⟨ Add more rows to a, until 12-bit entries are obtained 213 ⟩ ≡
  repeat ⟨ Put the bits for the next row, times l, into a 214 ⟩;
    l ← l + l; decr(y);
  until l = 4096;
```

This code is used in section 218.

214. As before, *cur_gf* will contain the first `GF` command that has not yet been interpreted.

```
⟨ Put the bits for the next row, times l, into a 214 ⟩ ≡
  if blank_rows > 0 then decr(blank_rows)
  else if cur_gf ≠ eoc then
      begin x ← z;
      if starting_col > x then starting_col ← x;
      ⟨ Read and process GF commands until coming to the end of this row 215 ⟩;
      end;
```

This code is used in section 213.

215.

```
define do_skip ≡ z ← 0; paint_black ← false
  define end_with(#) ≡
          begin #; cur_gf ← get_byte; goto done1; end
  define five_cases(#) ≡ #, # + 1, # + 2, # + 3, # + 4
  define eight_cases(#) ≡ #, # + 1, # + 2, # + 3, # + 4, # + 5, # + 6, # + 7
  define thirty_two_cases(#) ≡ eight_cases(#), eight_cases(# + 8), eight_cases(# + 16), eight_cases(# + 24)
  define sixty_four_cases(#) ≡ thirty_two_cases(#), thirty_two_cases(# + 32)
⟨ Read and process GF commands until coming to the end of this row 215 ⟩ ≡
  loop begin continue: case cur_gf of
    sixty_four_cases(0): k ← cur_gf;
    paint1: k ← get_byte;
    paint2: k ← get_two_bytes;
    paint3: k ← get_three_bytes;
    eoc: goto done1;
    skip0: end_with(blank_rows ← 0; do_skip);
    skip1: end_with(blank_rows ← get_byte; do_skip);
    skip2: end_with(blank_rows ← get_two_bytes; do_skip);
    skip3: end_with(blank_rows ← get_three_bytes; do_skip);
    sixty_four_cases(new_row_0), sixty_four_cases(new_row_0 + 64), thirty_two_cases(new_row_0 + 128),
        five_cases(new_row_0 + 160): end_with(z ← cur_gf − new_row_0; paint_black ← true);
    xxx1, xxx2, xxx3, xxx4, yyy, no_op: begin skip_nop; goto continue;
      end;
    othercases bad_gf('Improper␣opcode')
    endcases;
    ⟨ Paint k bits and read another command 216 ⟩;
    end;
done1:
```

This code is used in section 214.

216. ⟨ Paint k bits and read another command 216 ⟩ ≡

```
if x + k > finishing_col then finishing_col ← x + k;
if paint_black then
  for j ← x to x + k − 1 do a[j] ← a[j] + l;
paint_black ← ¬paint_black; x ← x + k; cur_gf ← get_byte
```

This code is used in section 215.

217. When the current row has been typeset, all entries of a will be even; we want to divide them by 2 and incorporate a new row with $l = 2^{11}$. However, if they are all multiples of 4, we actually want to divide by 4 and incorporate two new rows, with $l = 2^{10}$ and $l = 2^{11}$. In general, we want to divide by the maximum possible power of 2 and add the corresponding number of new rows; that's where the *rho* array comes in handy:

⟨ Advance to the next row that needs to be typeset; or **return**, if we're all done 217 ⟩ ≡

```
l ← rho[a[starting_col]];
for j ← starting_col + 1 to finishing_col do
  if l > rho[a[j]] then l ← rho[a[j]];
if l = 4096 then
  if cur_gf = eoc then return
  else begin y ← y − blank_rows; blank_rows ← 0; l ← 1; starting_col ← z; finishing_col ← z;
    end
else begin while a[starting_col] = 0 do incr(starting_col);
  while a[finishing_col] = 0 do decr(finishing_col);
  for j ← starting_col to finishing_col do a[j] ← a[j] div l;
  l ← 4096 div l;
  end
```

This code is used in section 218.

218. We now have constructed the major components of the necessary routine; it simply remains to glue them all together in the proper framework.

```
procedure do_pixels;
  label done, done1, reswitch, continue, exit;
  var paint_black: boolean;  { the paint switch }
    starting_col, finishing_col: 0 .. widest_row;  { currently nonzero area }
    j: 0 .. widest_row;  { for traversing that area }
    l: integer;  { power of two used to manipulate bit patterns }
    i: four_quarters;  { character information word }
    v: eight_bits;  { character corresponding to a pixel pattern }
  begin select_font(gray_font); delta_x ← delta_x + round(unsc_x_ratio * min_x);
  for j ← 0 to max_x − min_x do a[j] ← 0;
  l ← 1; z ← 0; starting_col ← 0; finishing_col ← 0; y ← max_y + 12; paint_black ← false;
  blank_rows ← 0; cur_gf ← get_byte;
  loop begin ⟨ Add more rows to a, until 12-bit entries are obtained 213 ⟩;
    dvi_goto(0, delta_y − round(unsc_y_ratio * y)); ⟨ Typeset the pixels of the current row 208 ⟩;
    dvi_out(pop); ⟨ Advance to the next row that needs to be typeset; or return, if we're all done 217 ⟩;
    end;
exit: end;
```

219. The main program. Now we are ready to put it all together. This is where `GFtoDVI` starts, and where it ends.

```
  begin initialize; { get all variables initialized }
  ⟨ Initialize the strings 77 ⟩;
  start_gf; { open the input and output files }
  ⟨ Process the preamble 221 ⟩;
  cur_gf ← get_byte; init_str_ptr ← str_ptr;
  loop begin ⟨ Initialize variables for the next character 144 ⟩;
    while (cur_gf ≥ xxx1) ∧ (cur_gf ≤ no_op) do ⟨ Process a no-op command 154 ⟩;
    if cur_gf = post then ⟨ Finish the DVI file and goto final_end 115 ⟩;
    if cur_gf ≠ boc then
      if cur_gf ≠ boc1 then abort('Missing␣boc!');
    ⟨ Process a character 164 ⟩;
    cur_gf ← get_byte; str_ptr ← init_str_ptr; pool_ptr ← str_start[str_ptr];
    end;
final_end: end.
```

220. The main program needs a few global variables in order to do its work.

```
⟨ Globals in the outer block 12 ⟩ +≡
k, m, p, q, r, s, t, dx, dy: integer; { general purpose registers }
time_stamp: str_number; { the date and time when the input file was made }
use_logo: boolean; { should METAFONT's logo be put on the title line? }
```

221. METAFONT sets the opening string to 32 bytes that give date and time as follows:

```
'␣METAFONT␣output␣yyyy.mm.dd:tttt'
```

We copy this to the `DVI` file, but remove the '`METAFONT`' part so that it can be replaced by its proper logo.

```
⟨ Process the preamble 221 ⟩ ≡
  if get_byte ≠ pre then bad_gf('No␣preamble');
  if get_byte ≠ gf_id_byte then bad_gf('Wrong␣ID');
  k ← get_byte; { k is the length of the initial string to be copied }
  for m ← 1 to k do append_char(get_byte);
  dvi_out(pre); dvi_out(dvi_id_byte); { output the preamble }
  dvi_four(25400000); dvi_four(473628672); { conversion ratio for sp }
  dvi_four(1000); { magnification factor }
  dvi_out(k); use_logo ← false; s ← str_start[str_ptr];
  for m ← 1 to k do dvi_out(str_pool[s + m − 1]);
  if str_pool[s] = "␣" then
    if str_pool[s + 1] = "M" then
      if str_pool[s + 2] = "E" then
        if str_pool[s + 3] = "T" then
          if str_pool[s + 4] = "A" then
            if str_pool[s + 5] = "F" then
              if str_pool[s + 6] = "O" then
                if str_pool[s + 7] = "N" then
                  if str_pool[s + 8] = "T" then
                    begin incr(str_ptr); str_start[str_ptr] ← s + 9; use_logo ← true;
                    end; { we will substitute 'METAFONT' for METAFONT }
  time_stamp ← make_string
```

This code is used in section 219.

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

223. Index. Here is a list of the section numbers where each identifier is used. Cross references to error messages and a few other tidbits of information also appear.