May 1989 Report No. STAN-CS-89-1255

METAFONTware

by

Donald E. Knuth, Tomas G. Rokicki, and Arthur L. Samuel

Department of Computer Science

Stanford University

Stanford, California 94305

The GFtype processor

(Version 3.0, April 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 GFtype utility program reads binary generic-font ("GF") files that are produced by font compilers such as METAFONT, and converts them into symbolic form. This program has three chief purposes: (1) It can be used to look at the pixels of a font, with one pixel per character in a text file; (2) it can be used to determine whether a GF file is valid or invalid, when diagnosing compiler errors: and (3) it serves as an example of a program that reads GF files correctly, for system programmers who are developing GF-related software.

The original version of this program was written by David R. Fuchs in March, 1984. Donald E. Knuth made a few modifications later that year as METAFONT was taking shape.

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

```
define banner ≡ 'This␣is␣GFtype,␣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, one extension is to use a default case as in TANGLE, WEAVE, etc. All places where nonstandard constructions are used have been listed in the index under "system dependencies."

```
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 binary input comes from *gf_file*, and the symbolic output is written on Pascal's standard *output* file. The term *print* is used instead of *write* when this program writes on *output*. so that all such output could easily be redirected if desired.

```
define print(#) ≡ write(#)
define print-ln(#) ≡ write_ln(#)
define print_nl ≡ write_ln
program GF_type(gf_file, output);
  label ⟨ Labels in the outer block 4⟩
  const ⟨ Constants in the outer block 5⟩
  type ⟨ Types in the outer block 8⟩
  var ⟨ Globals in the outer block 10⟩
  procedure initialize;   { this procedure gets things started properly }
    var i: integer;   { loop index for initializations }
    begin print-ln(banner);
    ⟨ Set initial values 11⟩
    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. Four parameters can be changed at compile time to extend or reduce GFtype's capacity. Note that the total number of bits in the main *image-array* will be

$$(max\text{-}row + 1) \times (max\_col + 1).$$

(METAFONT's full pixel range is rarely implemented, because it would require 8 megabytes of memory.)

```
⟨ Constants in the outer block 5⟩ ≡
  terminal_line_length = 150;
      { maximum number of characters input in a single line of input from the terminal }
  line-length = 79; { xxx strings will not produce lines longer than this }
  max-row = 79;   { vertical extent of pixel image array }
  max_col = 79;   { horizontal extent of pixel image array }
```

This code is used in section 3.

6. Here are some macros for common programming idioms.

```
  define incr (#) ≡ # ← # + 1    { increase a variable by unity }
  define decr (#) ≡ # ← # − 1    { decrease a variable by unity }
  define negate(#) ≡ # ← −#      { change the sign of a variable }
```

7. If the GF file is badly malformed, the whole process must be aborted; GFtype 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 GFtype.

```
  define abort (#) ≡
          begin print ( ´␣´, #); jump-out;
          end
  define bad_gf (#) ≡ abort( ´Bad␣GF␣file:␣´, #, ´!´)
procedure jump-out;
  begin goto final-end;
  end;
```

8. **The character set.** Like all programs written with the WEB system, GFtype 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.

The next few sections of GFtype have therefore been copied from the analogous ones in the WEB system routines. They have been considerably simplified, since GFtype need not deal with the controversial ASCII codes less than ´40. If such codes appear in the GF file, they will be printed as question marks.

⟨ Types in the outer block 8 ⟩ ≡
ASCII-code = "␣" . . "~"; { a subrange of the integers }

See also sections 9, 20, and 36.

This code is used in section 3.

9. The original Pascal compiler was designed in the late 60s, when six-bit character sets were common, so it did not make provision for lower case letters. Nowadays, of course, we need to deal with both upper and lower case alphabets in a convenient way, especially in a program like GFtype. So we shall assume that the Pascal system being used for GFtype has a character set containing at least the standard visible characters of ASCII code ("!" through "~").

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 (lust-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 *lust-text-char* = 127 { ordinal number of the largest element of *text-char* }

⟨ Types in the outer block 8 ⟩ +≡
text-file = packed file of *text-char;*

10. The GFtype 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 10 ⟩ ≡
xord: array *[text-char]* of *ASCII-code;* { specifies conversion of input characters }
xchr: array [0 . . 255] of *text-char;* { specifies conversion of output characters }

See also sections 21, 23, 25, 27, 35, 37, 39, 41, 46, 54, 62, and 67.

This code is used in section 3.

11. 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 11⟩ ≡
  for i ← 0 to '37 do xchr[i] ← '?';
  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] ← '~';
  for i ← '177 to 255 do xchr[i] ← '?';
```

See also sections 12, 26, 47, and 63.

This code is used in section 3.

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

```
⟨ Set initial values 11⟩ +≡
  for i ← first_text_char to last_text_char do xord[chr(i)] ← '40;
  for i ← "␣" to "~" do xord[xchr[i]] ← i;
```

13. Generic font file format. The most important output produced by a typical run of METAFONT is the "generic font" (GF) file that specifies the bit pat terns of the characters that have been drawn. 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 file formats have a lot in common. It is therefore not surprising that GFtype is identical in many respects to the DVItype program.

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 TFM 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 nurnber 0, then comes number 1, and so on.

14. 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.

15. 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-l 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;* METRFONT uses it to paint when $64 \leq d < 256$.

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] *mux-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* $\leq$ *m* $\leq$ *mux-m* and *min_n* $\leq$ *n* $\leq$ *mux-n*. (The values of *mux-m* and *min-n* need not be the tightest bounds possible.) When a GF-reading program sees a *boc*, it can use min-m, mux-m, min-n, and mux-n to initialize the bounds of an array. Then it sets m ← min-m, n ← *mux-n,* and *paint-switch* ← *white.*

boc1 6 8 $c[1]$ *del_m*[1] *max_m*[1] *del_n*[1] *mux-n* [1]. Same as *boc*, but *p* is assumed to be -1; also *del-m* = *max_m* − *min-m* and *del-n* = *mux-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* ← *min-m, paint-switch* ← *white.* (This finishes one row and begins another, ready to whiten the leftmost pixel in the new row.)

skip1 71 d[l]. Decrease *n* by $d + 1$, set *m* ← *min-m,* and set *paint-switch* ← *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-O 74. Decrease *n* by 1 and set *m* ← **min_m**, *paint-switch* ← *black.* (This finishes one row and begins another, ready to blacken the leftmost pixel in the new row.)

new_row_1 through *new-row-l 64* (opcodes 75 to 238). Same as *new-row-o,* but with *m* ← *min-m* + 1 through *min-m* + 164, respectively.

xxx1 2 3 9 $k[1]$ $x[k]$. 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 anywhere 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 $0 \leq k < 65536$.

xxx3 241 $k[3]$ $x[k]$. Like *xxx1*, but $0 \leq k < 2^{24}$. METRFONT 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-Zoc 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[l]* *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 }

16. Here are the opcodes that **GFtype** 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 *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 *skip 1* = 71 { skip over blank rows }
define *new-row-0* = 74 { move down one row and then right }
define *xxx1* = 239 { for special strings }
define *yyy* = 243 { for numspecial numbers }
define *no-op* = 244 { no operation }
define *char_loc* = 245 { character locators in the postamble }
define *pre* = 247 { preamble }
define *post* = 248 { postamble beginning }
define *post-post* = 249 { postamble ending }
define *undefined-commands* ≡ **250, 251, 252, 253, 254, 255**

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

> *post* $p[4]$ $ds[4]$ $cs[4]$ $hppp[4]$ $vppp[4]$ $min\_m[4]$ $mux\text{-}m[4]$ $min\_n[4]$ $max\_n[4]$
> ⟨ character locators ⟩
> *post-post* $q[4]$ $i[1]$ 223's$[\geq 4]$

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 postarnble. The *ds* and cs parameters give the design size and check sum, respectively, which are exactly the values put into the header of any TFM file that shares information with this GF file. 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, *mux-m,* min-n, and *mux-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; *mux-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 $\leq$ -50. If there are no characters in the file, it's possible to have *min-m* > *mux-m* and/ or *min-n* > *mux-n* .)

18. Character locators are introduced by *char-Zoc* 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^{24} 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.

19. 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., "DF in hexadecimal). METRFONT 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.

20. Input from binary files. We have seen that a GF file is a sequence 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 GFtype is written in standard Pascal.

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 8 ⟩ +≡
  eight-bits = 0 . . 255;  { unsigned one-byte quantity }
  byte-file = packed file of eight-bits;  { files that contain binary data }
```

21. The program deals with one binary file variable: *gf_file* is the main input file that we are translating into symbolic form.

```
⟨ Globals in the outer block 10 ⟩ +≡
gf-file: byte-file;  { the stuff we are GFtyping }
```

22. To prepare this file for input, we *reset* it.

```
procedure open_gf_file;  { prepares to read packed bytes in gf-file }
  begin reset (gf-file ); cur-Zoc ← 0;
  end;
```

23. If you looked carefully at the preceding code, you probably asked, "What is *cur_loc*?" Good question. It's a global variable that holds the number of the byte about to be read next from *gf-file*.

```
⟨ Globals in the outer block 10 ⟩ +≡
cur_loc: integer;  { where we are about to look, in gf_file }
```

24. We shall use a 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-Zoc); 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-Zoc ← cur-Zoc + 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-Zoc ← cur-Zoc + 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-Zoc + 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:
```

25. Optional modes of output. GFtype will print different quantities of information based on some options that the user must specify: We set *wants-mnemonics* if the user wants to see a mnemonic dump of the GF file; and *we* set *wants-pixels* if the user wants to see a pixel image of each character.

When GFtype begins, it engages the user in a brief dialog so that the options will be specified. This part of GFtype requires nonstandard Pascal constructions to handle the online interaction: so it may be preferable in some cases to omit the dialog and simply to produce the maximum possible output *(wants-mnemonics = wants_pixels = true).* On other hand, the necessary system-dependent routines are not complicated, so they can be introduced without terrible trauma.

```
⟨ Globals in the outer block 10⟩ +≡
wants-mnemonics: boolean;  { controls mnemonic output }
wants-pixels: boolean;  { controls pixel output }
```

26. ⟨ Set initial values 11⟩ +≡

```
  wants_mnemonics ← true; wants-pixels ← true;
```

27. The *input-Zn* 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, and *term-out* for terminal output.

```
⟨ Globals in the outer block 10⟩ +≡
buffer: array [0 . . terminal-line-length] of ASCII-code;
term-in: text-file;  { the terminal, considered as an input file }
term-out: text-file;  { the terminal, considered as an output file }
```

28. 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 invoke a system-dependent subroutine *update-terminal* in order to avoid this problem.

```
  define update-terminal ≡ break (term-out)  { empty the terminal output buffer }
```

29. During the dialog, extensions of GFtype might treat the first blank space in a line as the end of that line. Therefore *input-Zn* makes sure that there is always at least one blank space in *buffer*.

(This routine is more complex than the present implementation needs, but it has been copied from DVItype so that system-dependent changes that worked before will work again.)

```
procedure inpu t-Zn ;  { inputs a line from the terminal }
  var k: 0.. terminal_line_length;
  begin update-terminal; reset (term-in);
  if eoln ( term-in) then read-Zn ( term-in);
  k ← 0:
  while (k < terminal-line-length) A ¬eoln( term-in) do
    begin buffer [k] ←xord [ term-in ↑]; incr (k); get ( term-in);
    end;
  buffer[k] ← "␣";
  end:
```

30. This is humdrum.

```
function lower_casify(c : ASCII-code): ASCII-code;
  begin if (c ≥ "A") A (c ≤ "Z") then lower-cusify ← c + "a" − "A"
  else lower-cusify ← c;
  end;
```

31. The selected options are put into global variables by the *dialog* procedure, which is called just, as GFtype begins.

```
procedure dialog;
  label 1, 2;
  begin rewrite (term-out );  { prepare the terminal for output }
  write-Zn (term-out, banner);
  ⟨Determine whether the user wants-mnemonics 32⟩;
  ⟨Determine whether the user wants_pixels 33⟩;
  ⟨Print all the selected options 34⟩:
  end;
```

32. ⟨Determine whether the user *wants-mnemonics* 32⟩ ≡

```
1: write (term-out, ´Mnemonic␣output?␣(default=no,␣?␣for␣help):␣´); input_ln;
  buffer [0]← lower_casify (buffer [0]);
  if buffer[0] ≠ "?" then wants-mnemonics ← (buffer [0] = "y") ∨ (buffer [0] = "1") ∨ (buffer [0] = "t")
  else begin write (term-out, ´Type␣Y␣for␣complete␣listing,´);
    write-Zn( term-out, ´␣N␣for␣errors/images␣only.´); goto 1;
    end
```

This code is used in section 31.

33. ⟨Determine whether the user *wants-pixels* 33⟩ ≡

```
2: write(term_out, ´Pixel␣output?␣(default=yes,␣?␣for␣help):␣´); input-Zn;
  buffer [0]← lower-cusify (buffer [0]);
  if buffer[0] ≠ "?" then
    wants-pixels ← (buffer[0] = "y") ∨ (buffer[0] = "1") ∨ (buffer[0] = "t") ∨ (buffer[0] = "␣")
  else begin write ( term-out, ´Type␣Y␣to␣list␣characters␣pictorially´);
    write_ln(term_out, ´␣with␣*´´s,␣N␣to␣omit␣this␣option.´); goto 2;
    end
```

This code is used in section 31.

34. After the dialog is over, we print the options so that the user can see what, GFtype thought was specified.

⟨Print all the selected options 34⟩ ≡

```
  print(´Options␣selected:␣Mnemonic␣output␣=␣´);
  if wants-mnemonics then print (´true´) else print (´false´);
  print (´;␣pixel␣output␣=␣´);
  if wants-pixels then print (´true´) else print (´false´);
  print-Zn(´.´)
```

This code is used in section 31.

35. The image array. The definition of GF files refers to two registers, m and n, which hold integer column and row numbers. We actually keep the values $m' = m - min\_m$ and $n' = max\_n - n$ instead, so that our internal image array always has $m, n \geq 0$. We also need to remember *paint-switch,* whose value is either *black* or *white.*

```
⟨ Globals in the outer block 10 ⟩ +≡
m, n: integer;   { current state values, modified by min-m and max-n }
paint-switch: pixel;
```

36. We'll need a big array of pixels to hold the character image. Each pixel should be represented as a single bit in order to save space. Some systems may prefer the following definitions, while others may do better using the *boolean* type and boolean constants.

```
  define white = 0   { could also be false }
  define black = 1   { could also be true }
⟨ Types in the outer block 8 ⟩ +≡
  pixel = white . . black;   { could also be boolean }
```

37. In order to allow different systems to change the *image* array easily from row-major order to column-major order (or vice versa), or to transpose it top and bottom or left and right, we declare and access it as follows.

```
  define image ≡ image-array [m, n]
⟨ Globals in the outer block 10 ⟩ +≡
image-array: packed array [0 . . max-col, 0 . . max_row] of pixel;
```

38. A *boc* command has parameters *min_m*, *max_m*, *min_n*, and max-n that define a rectangular subarray in which the pixels of the current character must lie. The program here computes limits on GFtype's modified m and n variables, and clears the resulting subarray to all *white.*

(There may be a faster way to clear a subarray on particular systems, using nonstandard extensions of Pascal.)

```
⟨ Clear the image 38 ⟩ ≡
  begin max-subcol ← max_m_stated − min-m-stated − 1;
  if max-subcol > max_col then max-subcol ← max-col;
  maxsubrow ← max_n_stated − min-n-stated;
  if max_subrow > max-row then max_subrow ← max-row;
  n ← 0;
  while n ≤ max-subrow do
    begin m ← 0;
    while m ≤ max-subcol do
      begin image ← white: incr(m);
      end:
    incr (n);
    end:
  end
```

This code is used in section 71.

39. ⟨ Globals in the outer block 10 ⟩ +≡

```
max_subrow, max-subcol: integer;   { size of current subarray of interest }
```

40. As we paint the pixels of a character, we will record its actual boundaries in variables *max_m_observed* and *max_n_observed*. Then the following routine will be called on to output the image, using blanks for *white* and asterisks for *black*. Blanks are emitted only when they are followed by nonblanks, in order to conserve space in the output. Further compaction could be achieved on many systems by using tab marks.
An integer variable *b* will be declared for use in counting blanks.

```
⟨ Print the image 40 ⟩ ≡
  begin ⟨ Compare the subarray boundaries with the observed boundaries 42 ⟩;
  if max_subcol ≥ 0 then   { there was at least one paint command }
    ⟨ Print asterisk patterns for rows 0 to max_subrow 43 ⟩
  else print_ln(´(The␣character␣is␣entirely␣blank.)´);
  end
```

This code is used in section 69.

41. ⟨ Globals in the outer block 10 ⟩ +≡

```
min_m_stated, max_m_stated, min_n_stated, max_n_stated: integer;   { bounds stated in the GF file }
max_m_observed, max_n_observed: integer;   { bounds on (m′, n′) actually observed when painting }
min_m_overall, max_m_overall, min_n_overall, max_n_overall: integer;
      { bounds observed in the entire file so far }
```

42. If the given character is substantially smaller than the *boc* command predicted, we don't want to bother to output rows and columns that are all blank.

```
⟨ Compare the subarray boundaries with the observed boundaries 42 ⟩ ≡
  if (max_m_observed > max_col) ∨ (max_n_observed > max_row) then
    print_ln(´(The␣character␣is␣too␣large␣to␣be␣displayed␣in␣full.)´);
  if max_subcol > max_m_observed then max_subcol ← max_m_observed;
  if max_subrow > max_n_observed then max_subrow ← max_n_observed;
```

This code is used in section 40.

43. ⟨ Print asterisk patterns for rows 0 to max_subrow 43 ⟩ ≡

```
  begin print_ln(´.<--This␣pixel´´s␣lower␣left␣corner␣is␣at␣(´, min_m_stated : 1, ´,´,
      max_n_stated + 1 : 1, ´)␣in␣METAFONT␣coordinates´); n ← 0;
  while n ≤ max_subrow do
    begin m ← 0; b ← 0;
    while m ≤ max_subcol do
      begin if image = white then incr(b)
      else begin while b > 0 do
          begin print(´␣´); decr(b);
          end;
        print(´*´);
        end;
      incr(m);
      end;
    print_nl; incr(n);
    end;
  print_ln(´.<--This␣pixel´´s␣upper␣left␣corner␣is␣at␣(´, min_m_stated : 1, ´,´,
      max_n_stated − max_subrow : 1, ´)␣in␣METAFONT␣coordinates´);
  end
```

This code is used in section 40.

44. Translation to symbolic form. The main work of **GFtype** is accomplished by the *do-char* procedure, which produces the output for an entire character, assuming that the *boc* command for that character has already been processed. This procedure is essentially an interpretive routine that reads and acts on the GF commands.

45. We steal the following routine from METAFONT.

```
  define unity ≡ '200000   { $2^{16}$, represents 1.00000)
procedure print-scaled (s : integer);  {prints a scaled number, rounded to five digits }
  var delta: integer;  { amount of allowable inaccuracy }
  begin if s < 0 then
    begin print ('-'); negate(s);  {print the sign, if negative}
    end;
  print(s div unity : 1);  { print the integer part }
  s ← 10 * (s mod unity) + 5;
  if s ≠ 5 then
    begin delta + 10; print('.');
    repeat if delta > unity then s ← s + '100000 − (delta div 2);  { round the final digit }
      print (chr (ord ('0') + (s div unity))); s ← 10 * (s mod unity); delta ← delta * 10;
    until s ≤ delta;
    end;
  end;
```

46. Let's keep track of how many characters are in the font, and the locations of where each one occurred in the file.

```
⟨ Globals in the outer block 10 ⟩ +≡
total-chars: integer;  { the total number of characters seen so far }
char-ptr: array [0 . . 255] of integer;  { correct character location pointer}
gf_prev_ptr: integer;  { char-ptr for next character}
character-code: integer;  { current character number }
```

47. (Set initial values 11) +≡

```
  for i ← 0 to 255 do char-ptr[i] ← -1;  { mark characters as not being in the file }
  total-chars ← 0;
```

48. Before we get into the details of *do-char,* it is convenient to consider a simpler routine that computes the first parameter of each opcode.

```
  define four-cases (#) ≡ #, # + 1, # + 2, # + 3
  define eight-cases (#) ≡ four-cases (#), four-cases (# + 4)
  define sixteen-cases (#) ≡ eight-cases (#), eight-cases (# + 8)
  define thirty-two-cases (#) ≡ sixteen-cases (#), sixteen-cases (# + 16)
  define thirty-seven-cases (#) ≡ thirty-two-cases (#), four-cases (# + 32). # + 36
  define sixty-four-cases (#) ≡ thirty-two-cases (#), thirty-two-cases (# + 32)
function first-par(o : eight-bits): integer:
  begin case o of
  sixty-four-cases (paint-o): first-par + 0 − paint-O;
  paint1 , skip1 , char-lot, char-lot + 1, xxx1: first-par + get-byte;
  paint1 + 1, skip1 + 1, xxx1 + 1: first-par ← get-two-bytes;
  paint1 + 2, skip1 + 2, xxx1 + 2: first-par + get-three-bytes;
  xxx1 + 3, yyy: first-par ← signed-quad;
  boc, boc1 , eoc, skip0, no_op, pre, post, post-post, undefined-commands: first-par ← 0:
  sixty_four_cases( new-row-o, sixty-four-cases (new-row-0 + 64), thirty-seven-cases (new-row-O + 128):
        first-par + 0 − new-row-O;
  end;
  end;
```

49. Strictly speaking, the *do-char* procedure is really a function with side effects, not a 'procedure' ; it returns the value *false* if **GFtype** should be aborted because of some unusual happening. The subroutine is organized as a typical interpreter, with a multiway branch on the command code.

```
function do-char: boolean ;
  label 9998, 9999;
  var o: eight-bits;  { operation code of the current command }
    p, q: integer;  { parameters of the current command }
    aok: boolean;  { the value to return }
  begin  { we've already scanned the boc }
  aok ← true;
  while true do ⟨ Translate the next command in the GF file; goto 9999 if it was eoc; goto 9998 if
        premature termination is needed 50⟩;
9998: print-ln( ´ ! ´); aok + false;
9999: do-char ← aok;
  end;
```

```
50.  define show-label(#) ≡ print(a : 1, ´:␣´, #)
  define show-mnemonic (#) ≡
          if wants-mnemonics then
            begin print-nl: show-label(#);
            end
  define error(#) ≡
          begin show-label ( ´ ! ␣´, #); print-nl:
          end
  define nl_error (#) ≡
          begin print-nl; show_label( ´ ! ␣´, #); print-nl;
          end
  define start-op ≡ a ← cur_loc; o ← get-byte; p ← first-par(o);
        if eof (gf_file) then bad-gf( ´the␣f ile␣ended␣prematurely´)
⟨ Translate the next command in the GF file; goto 9999 if it was eoc; goto 9998 if premature termination is
      needed 50⟩ ≡
  begin start-op; ⟨ Start translation of command o and goto the appropriate label to finish the job 51⟩;
  end
```

This code is used in section 49.

51. The multiway switch in *first-par,* above, was organized by the length of each command; the one in *do-char* is organized by the semantics.

```
⟨ Start translation of command o and goto the appropriate label to finish the job 51⟩ ≡
  if o ≤ paint1 + 3 then ⟨ Translate a sequence of paint commands, until reaching a non-paint 56⟩;
  case 0 of
  four-cases(skip0): ⟨ Translate a skip command 60⟩;
  sixty-four-cases (new_row_0), sixty-four-cases (new-row-0 + 64), thirty-seven-cases (new-row-0 + 128):
        ⟨ Translate a new-row command 59⟩;
  ⟨ Cases for commands no-op, pre, post, post-post, boc, and eoc 52 ⟩
  four-cases (xxx1 ): ⟨ Translate an xxx command 53⟩;
  yyy: ⟨Translate a yyy command 55⟩;
  othercases error( ´undef ined␣command␣´, o : 1, ´ !´)
  endcases
```

This code is used in section 50.

```
52.  ⟨ Cases for commands no-op, pre, post, post-post, boc, and eoc 52⟩ ≡
no-op: show-mnemonic( ´no␣op´);
pre: begin error( ´preamble␣command␣within␣a␣character!´); goto 9998;
  end;
post, post-post: begin error(´postamble␣command␣within␣a␣character!´); goto 9998;
  end;
boc, boc1 : begin error( ´boc␣occurred␣before␣eoc!´); goto 9998;
  end;
eoc: begin show-mnemonic( ´eoc´); print_nl; goto 9999:
  end;
```

This code is used in section 51.

53. ⟨ Translate an xxx command 53 ⟩ ≡

```
begin show_mnemonic('xxx␣'''); bad-char ← false; b ← 16;
if p < 0 then nl_error('string␣of␣negative␣length!');
while p > 0 do
  begin q ← get-byte;
  if (q < "␣") ∨ (q > "~") then bad-char ← true;
  if wants-mnemonics then
    begin print (xchr [q]);
    if b < line-length then incr(b)
    else begin print-m; b ← 2:
      end;
    end;
  decr(p);
  end;
if wants-mnemonics then print ('''');
if bad-char then nl_error('non-ASCII␣character␣in␣xxx␣command!');
end
```

This code is used in sections 51 and 70.

54. ⟨ Globals in the outer block 10 ⟩ +≡
bad-char: boolean; { has a non-ASCII character code appeared in this xxx? }

55. ⟨ Translate a yyy command 55 ⟩ ≡

```
begin show-mnemonic ('yyy␣', p : 1, '␣(');
if wants-mnemonics then
  begin print-scaled(p); print (')');
  end;
end
```

This code is used in sections 51 and 70.

56. The bulk of a GF file generally consists of *paint* commands, so we collect them together and print them in an abbreviated format on one line.

⟨ Translate a sequence of *paint* commands, until reaching a *non-paint* 56 ⟩ ≡

```
begin if wants-mnemonics then print ('␣paint␣');
repeat ⟨ Paint the next p pixels 57 ⟩;
  start-op;
until 0 > paint1 + 3;
end
```

This code is used in section 51.

57. ⟨ Paint the next *p* pixels 57 ⟩ ≡

```
if wants-mnemonics then
  if paint-switch = white then print('(', p : 1, ')') else print(p: 1);
m ← m + p;
if m > max-m-observed then max_m_observed ← m − 1;
if wants-pixels then ⟨ Paint pixels m − p through m − 1 in row n of the subarray 58 ⟩;
paint-switch ← white + black − paint-switch   { could also be paint_switch ← -paint-switch }
```

This code is used in section 56.

58. We use the fact that the subarray has been initialized to all *white*.

```
⟨ Paint pixels $m - p$ through $m - 1$ in row $n$ of the subarray 58 ⟩ ≡
  if paint-switch = black then
    if $n \le$ max_subrow then
      begin $l \leftarrow m - p$; $r \leftarrow m - 1$;
      if $r >$ max_subcol then $r \leftarrow$ max_subcol;
      m ← $l$;
      while $m \le r$ do
        begin image ← black; incr(m);
        end:
      $m \leftarrow l + p$;
      end
```

This code is used in section 57.

59. ⟨ Translate a *new-row* command 59 ⟩ ≡

```
  begin show-mnemonic('newrow␣', p : 1); incr (n); m ← p; paint-switch ← black;
  if wants-mnemonics then print('␣(n=', max_n_stated − n : 1, ')');
  end
```

This code is used in section 51.

60. ⟨ Translate a *skip* command 60 ⟩ ≡

```
  begin show-mnemonic( 'skip', (o − skip1 + 1) mod 4 : 1, '␣', p : 1); n ← n + p + 1; m ← 0;
  paint-switch ← white;
  if wants-mnemonics then print('␣(n=', max-n-stated − n : 1, ')');
  end
```

This code is used in section 51.

61. **Reading the postamble.** Now imagine that we are reading the GF file and positioned just after the *post* command. That, in fact, is the situation, when the following part of GFtype is called upon to read, translate, and check the rest of the postamble.

```
procedure read-postamble;
  var k: integer;  {loop index}
    p, q, m, u, v, w, c: integer;  {general purpose registers}
  begin post-Zoc + cur-lot − 1; print ('Postamble␣starts␣at␣byte␣', post_loc : 1);
  if post-lot = gf-prev-ptr then print_ln('.')
  else print-ln (',␣after␣special␣info␣at␣byte␣', gf_prev_ptr : 1, '.');
  p ← signed-quad;
  if p ≠ gf-prev-ptr then
    error('backpointer␣in␣byte␣', cur_loc − 4 : 1, '␣should␣be␣', gf-prev-ptr : 1, '␣not␣', p : 1. '!');
  design-size + signed-quad; check-sum ← signed-quad;
  print ('design␣size␣=␣', design-size : 1, '␣('); print-scaled (design-size div 16); print_ln('pt)');
  print_ln ('check␣sum␣=␣', check-sum : 1);
  hppp + signed-quad; vppp ← signed-quad;
  print('hppp␣=␣', hppp : 1, '␣('); print-scaZed(hppp); print-ln(')'); print('vppp␣=␣', vppp : 1, '␣(');
  print-scaled (vppp); print-ln (')'); pix-ratio ← (design-size / 1048576) * (hppp / 1048576);
  min-m-stated ← signed-quad; max-m-stated + signed-quad; min-n-stated ← signed-quad;
  max-n-stated + signed-quad;
  print-Zn('min␣m␣=␣', min-m-stated : 1, ',␣max␣m␣=␣', max-m-stated : 1);
  if min-m-stated > min-m-overall then error('min␣m␣should␣be␣<=', min-m-overall : 1. '!');
  if max-m-stated < max-m-overall then error('max␣m␣should␣be␣>=', max-m-overall : 1. '!');
  print-Zn('min␣n␣=␣', min_n_stated : 1, ',␣max␣n␣=␣', max-n-stated : 1);
  if min-n-stated > min-n-overall then error('min␣n␣should␣be␣<=', min-n-overall : 1, '!');
  if max-n-stated < max-n-overall then error('max␣n␣should␣be␣>=', max-n-overall : 1, '!');
  ⟨ Process the character locations in the postamble 65⟩;
  ⟨ Make sure that the end of the file is well-formed 64⟩;
  end;
```

62. ⟨ Globals in the outer block 10⟩ +≡

```
design-size, check-sum: integer;  { TFM-oriented parameters }
hppp, vppp: integer;  {magnification-oriented parameters}
post-lot: integer;  {location of the post command}
pix-ratio: real;  {multiply by this to convert TFM width to scaled pixels}
```

63. ⟨ Set initial values 11⟩ +≡

```
  min-m-overall ← max-int ; max-m-overall ← − max-int ; min-n-overall + max_int ;
  rnax-n-overall + − max-int ;
```

64. When we get to the present code, the *post-post* command has just been read.

```
⟨ Make sure that the end of the file is well-formed 64⟩ ≡
  if k ≠ post-post then error('should␣be␣postpost!');
  q + signed-quad;
  if q ≠ post_loc then error('postamble␣pointer␣should␣be␣', post_loc : 1, '␣not␣', q : 1, '!');
  m + get-byte;
  if m ≠ gf-id-byte then error('identification␣byte␣should␣be␣', gf-id-byte : 1, ',␣not␣', m : 1, '!');
  k + cur_loc; m ← 223;
  while (m = 223) A ¬eof (gf_file) do m ← get-byte;
  if ¬eof (gf-file) then bad-gf('signature␣in␣byte␣', cur-Zoc − 1 : 1, '␣should␣be␣223')
  else if cur_loc < k + 4 then error('not␣enough␣signature␣bytes␣at␣end␣of␣f ile!');
```

This code is used in section 61.

65. ⟨ Process the character locations in the postamble 65⟩ ≡

```
repeat $a \leftarrow cur\text{-}lot$; $k \leftarrow get\text{-}byte$;
  if $(k = char\text{-}Zoc) \lor (k = char\text{-}lot + 1)$ then
    begin $c \leftarrow first\text{-}par(k)$;
    if $k = char\text{-}lot$ then
      begin $u \leftarrow signed\text{-}quad$; $v \leftarrow signed\text{-}quad$:
      end
    else begin $u \leftarrow get\text{-}byte * unity$; $v \leftarrow 0$:
      end;
    $w \leftarrow signed\text{-}quad$; $p \leftarrow signed\text{-}quad$; print(´Character␣´, $c$ : 1, ´:␣dx␣´, $u$ : 1, ´␣(´);
    print-scaled(u);
    if v $\neq 0$ then
      begin print(´),␣dy␣´, $v$ : 1, ´␣(´); print-scaled(v);
      end;
    print(´),␣width␣´, $w$ : 1, ´␣(´); $w \leftarrow round(w * pix\_ratio)$; print-scaled(w);
    print-ln(´),␣loc␣´, $p$ : 1);
    if $p \neq char\text{-}ptr[c]$ then error(´character␣location␣should␣be␣´, char-ptr[c] : 1, ´!´);
    $k \leftarrow no\text{-}op$;
    end;
until $k \neq no\text{-}op$;
```

This code is used in section 61.

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

```
  begin initialize;  { get all variables initialized }
  dialog;  { set up all the options }
  ⟨ Process the preamble 68 ⟩;
  ⟨ Translate all the characters 69⟩;
  print_nl; read-postamble; print (´The␣file␣had␣´, total-chars : 1, ´␣character´);
  if total-chars ≠ 1 then print(´s´);
  print(´␣altogether.´);
final-end: end.
```

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

```
⟨ Globals in the outer block 10⟩ +≡
a: integer;  { byte number of the current command }
b, c, l, o, p, q, r: integer;  { general purpose registers }
```

68. GFtype looks at the preamble in order to do error checking, and to display the introductory comment.

```
⟨ Process the preamble 68 ⟩ ≡
  open_gf_file; o ← get-byte;  { fetch the first byte }
  if o ≠ pre then bad-gf (´First␣byte␣isn´´t␣start␣of␣preamble!´);
  o ← get-byte;  { fetch the identification byte }
  if o ≠ gf-id-byte then bad-gf ( ‘identif ication␣byte␣should␣be␣´, gf-id-byte : 1, ´␣not␣´, o : 1);
  o ← get-byte;  { fetch the length of the introductory comment }
  print(´´´´);
  while o > 0 do
    begin decr(o); print(xchr[get_byte]);
    end;
  print-ln (´´´´);
```

This code is used in section 66.

69. ⟨ Translate all the characters 69⟩ ≡

```
  repeat gf-prev-ptr ← cur-Zoc; ⟨ Pass no-op, xxx and yyy commands 70⟩;
    if o ≠ post then
      begin if o ≠ boc then
        if o ≠ boc1 then bad-gf (´byte␣´, cur-Zoc − 1 : 1, ´␣is␣not␣boc␣(´, o : 1, ´)´);
      print_nl; print (cur-lot − 1 : 1, ´ : ␣beginning␣of␣char␣´); ⟨ Pass a boc command 71 ⟩;
      if ¬do_char then bad-gf (´char␣ended␣unexpectedly´);
      max_n_observed ← n;
      if wants_pixels then ⟨ Print the image 40 ⟩;
      ⟨ Pass an eoc command 72 ⟩;
      end:
  until 0 = post;
```

This code is used in section 66.

```
70. ⟨Pass no-op, xxx and yyy commands 70⟩ ≡
  repeat start_op;
    if o = yyy then
      begin ⟨Translate a yyy command 55⟩;
      o ← no-op;
      end
    else if (o ≥ xxx1) ∧ (o ≤ xxx1 + 3) then
        begin ⟨Translate an xxx command 53⟩;
        o ← no-op;
        end
      else if o = no-op then show-mnemonic('no␣op');
  until o ≠ no-op;
```

This code is used in section 69.

```
71. ⟨Pass a boc command 71⟩ ≡
  a ← cur_loc − 1; incr(total-chars);
  if o = boc then
    begin character-code ← signed-quad; p ← signed-quad; c ← character-code mod 256;
    if c < 0 then c ← c + 256;
    min-m-stated ← signed-quad; max-m-stated ← signed-quad; min-n-stated ← signed-quad;
    max-n-stated ← signed-quad;
    end
  else begin character-code ← get-byte; p ← -1; c ← character-code; q ← get-byte;
    max-m-stated ← get-byte; min-m-stated ← max-m-stated − q; q ← get-byte; max-n-stated ← get-byte;
    min-n-stated ← max-n-stated − q;
    end;
  print(c : 1);
  if character-code ≠ c then print('␣with␣extension␣', (character-code − c) div 256 : 1);
  if wants-mnemonics then print-ln(':␣', min-m-stated : 1, '<=m<=', max_m_stated : 1, '␣',
        min-n-stated : 1, '<=n<=', max-n-stated : 1);
  max_m_observed ← -1;
  if char-ptr[c] ≠ p then
    error('previous␣character␣pointer␣should␣be␣', char-ptr[c] : 1, ',␣not␣', p : 1, '!')
  else if p > 0 then
      if wants-mnemonics then
        print-ln('(previous␣character␣with␣the␣same␣code␣started␣at␣byte␣', p : 1, ')');
  char-ptr[c] ← gf-prev-ptr;
  if wants-mnemonics then print('(initially␣n=', max-n-stated : 1, ')');
  if wants-pixels then ⟨Clear the image 38⟩;
  m ← 0; n ← 0; paint-switch ← white;
```

This code is used in section 69.

72. ⟨Pass an eoc command 72⟩ ≡

```
max_m_observed ← min_m_stated + max_m_observed + 1; n ← max_n_stated − max_n_observed;
      { now n is the minimum n observed }
if min_m_stated < min_m_overall then min_m_overall ← min_m_stated;
if max_m_observed > max_m_overall then max_m_overall ← max_m_observed;
if n < min_n_overall then min_n_overall ← n;
if max_n_stated > max_n_overall then max_n_overall ← max_n_stated;
if max_m_observed > max_m_stated then
   print_ln(´The␣previous␣character␣should␣have␣had␣max␣m␣>=␣´, max_m_observed : 1, ´!´);
if n < min_n_stated then
   print_ln(´The␣previous␣character␣should␣have␣had␣min␣n␣<=␣´, n : 1, ´!´)
```

This code is used in section 69.

73. **System-dependent changes..** This section should be replaced, if necessary, by changes to the program that are necessary to make GFtype 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.

74. Index. Pointers to error messages appear here together with the section numbers where each identifier is used.

The GFtoPK processor

(Version 2.0, 17 April 1989)

The preparation of this report was supported in part by the National Science Foundation under grants ET-8201926, MCS-8300984, and CCR-8610181, and by the System Development Foundation. 'TEX' is a trademark of the American Mathematical Society. **'METRFONT'** is a trademark of Addison-Wesley Publishing Company.

1. Introduction. This program reads a GF file and packs it into a PK file. PK files are significantly smaller than GF files, and they are much easier to interpret. This program is meant to be the bridge between METRFONT and DVI drivers that read PK files. Here are some statistics comparing pixel files, compressed pixel files, generic font files, and packed files:

Font	Resolution	Pixel		Compressed		Generic font		Packed	
amr10	300	14928	70	9866	106	12768	82	5292	198
amr10	360	20100	52	13130	80	14816	71	6160	170
amr10	432	27556	38	18241	57	17704	59	7564	139
amr10	511	35652	29	25266	42	20712	51	9208	114
amr10	622	47324	22	36129	29	24396	43	11156	94
amr10	746	67108	16	50651	21	28848	36	13884	76
aminch	300	353432	3	329073	3	48900	21	22132	47
	(Set of 48)	792832	100%	516397	65%	589100	74%	261532	33%

(Pixel files represent an obsolete format that was used with METAFONT79.)

The first number in each column is the space required in bytes; the second number is the number of fonts of that size that would fit in one megabyte (1048576 bytes) of disk space, rounded to the nearest integer. The last row is the set of sixteen basic fonts at the three resolutions 300, 329, and 360, totaled, with the second number in each column being the percentage in bytes of the size of the set of files. The compressed pixel format is a hypothetical format similar to the old pixel files, but with the character rasters packed by the bit rather than by the 32-bit word; it represents a lower limit to the size of pixel files when a simple scheme like the standard pixel file format is used. The fact that the PK files for the set of 48 fonts is less than half the size of any of the other formats should indicate why this format is being introduced into a world where there are already too many font formats. It is hoped that the simplicity and small size of the PK files will make them widely accepted.

The PK format was designed and implemented by Tomas Rokicki during the summer of 1985. This program borrows a few routines from GFtoPXL by Arthur Samuel.

The *banner* string defined here should be changed whenever GFtoPK gets modified. The *preamble-comment* macro (near the end of the program) should be changed too.

define *banner* ≡ `'This␣is␣GFtoPK,␣Version␣2.0'` { printed when the program starts }

2. Some of the diagnostic information is printed using *d_print_ln*. When debugging, it should be set the same as *print_ln*, defined later.

define *d-print-ln* (#) ≡

3. This program is written in standard Pascal, except where it is necessary to use extensions; for example. one extension is to use a default case as in TANGLE, WEAVE, etc. All places where nonstandard constructions are used should be listed in the index under "system dependencies."

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*

4. The binary input comes from *gf_file*, and the output font is written on *pk_file*. All text output is written on Pascal's standard *output* file. The term *print* is used instead of *write* when this program writes on *output*. so that all such output could easily be redirected if desired.

```
  define print(#) ≡ write(#)
  define print-k (#) ≡ write_ln (#)
program GFtoPK (gf_file, pk_file, output );
  label ⟨ Labels in the outer block 5 ⟩
  const ⟨ Constants in the outer block 6⟩
  type ⟨ Types in the outer block 9 ⟩
  var ⟨ Globals in the outer block 11 ⟩
  procedure initialize ; { this procedure gets things started properly }
    var i: integer; { loop index for initializations }
    begin print_ln (banner);
    ⟨ Set initial values 12 ⟩
    end:
```

5. 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 5 ⟩ ≡
  final-end;
```

This code is used in section 4.

6. The following parameters can be changed at compile time to extend or reduce GFtoPK's capacity. The values given here should be quite adequate for most uses. Assuming an average of about three strokes per raster line, there are six run-counts per line, and therefore *max-row* will be sufficient for a character 2600 pixels high.

```
⟨ Constants in the outer block 6 ⟩ ≡
  line-length = 79; { bracketed lines of output will be at most this long }
  terminal-line-length = 150;
      { maximum number of characters input in a single line of input from the terminal }
  mux-row = 16000; { largest index in the main row array }
```

This code is used in section 4.

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 }
```

8. If the GF file is badly malformed, the whole process must be aborted; GFtoPK 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 GFtoPK.

```
  define abort (#) ≡
          begin print ('␣', #); jump-out;
          end
  define bad_gf (#) ≡ abort ('Bad␣GF␣f ile : ␣', #. '!')
procedure jump-out;
  begin goto final-end:
  end;
```

9. **The character set.** Like all programs written with the WEB system, GFtoPK 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.

The next few sections of GFtoPK have therefore been copied from the analogous ones in the WEB system routines. They have been considerably simplified, since GFtoPK need not deal with the controversial ASCII codes less than '40. If such codes appear in the GF file, they will be printed as question marks.

⟨ Types in the outer block 9 ⟩ ≡
  ASCII-code = "␣" . . "~"; { a subrange of the integers }

See also sections 10 and 37.

This code is used in section 4.

10. The original Pascal compiler was designed in the late 60s, when six-bit character sets were common, so it did not make provision for lower case letters. Nowadays, of course, we need to deal with both upper and lower case alphabets in a convenient way, especially in a program like GFtoPK. So we shall assume that the Pascal system being used for GFtoPK has a character set containing at least the standard visible characters of ASCII code ("!" through "~").

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* = 127 { ordinal number of the largest element of *text-char* }

⟨ Types in the outer block 9 ⟩ +≡
  text-file = **packed file of** *text-char*;

11. The GFtoPK 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 11 ⟩ ≡
xord: **array** [*text-char*] **of** *ASCII-code*; { specifies conversion of input characters }
xchr: **array** [0 . . 255] **of** *text-char*; { specifies conversion of output characters }

See also sections 38, 41, 45, 47, 48, 55, 78, 82, and 87.

This code is used in section 4.

12. 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 12⟩ ≡
  for i - 0 to ´37 do xchr[i] - ´?´;
  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] - ´~´;
  for i ← ´177 to 255 do xchr[i] ← ´?´;
```

See also sections 13, 42, 49, 79, and 83.

This code is used in section 4.

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

```
⟨ Set initial values 12⟩ +≡
  for i ← first-text-char to last-text-char do xord[chr(i)] ← ´40;
  for i - "␣" to "~" do xord[xchr[i]] ← i;
```

14. Generic font file format. The most important output produced by a typical run of METAFONT is the "generic font" (GF) file that specifies the bit patterns of the characters that have been drawn. 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 file formats have a lot in common.

A GF file is a stream of S-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 TFM 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.

15. 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 METRFONT? 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.

16. 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-69* (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]. Sarne 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* + *min-m, n* ← *mux-n,* and *paint-switch* + *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 ← min-m, *paint-switch* + *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* ← *min-m,* and set *paint-swatch* + *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-O 74. Decrease *n* by 1 and set *m* + *min-m, paint-switch* + *black.* (This finishes one row and begins another, ready to blacken the leftmost pixel in the new row.)

new-row-l through new-row-164 (opcodes 75 to 238). Same as *new-row-O,* but with *m* ← 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 anywhere 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. METRFONT 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-Zoc 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-Zoc,* 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 }

17. Here are the opcodes that GFtoPK 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 *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 *new-row-0* = 74 { move down one row and then right }
define *max_new_row* = 238 { move down one row and then right }
define *no-op* = *247* { noop }
define *xxx1* = 239 { for special strings }
define *yyy* = 243 { for numspecial numbers }
define *nop* = 244 { no operation}
define *char_loc* = 245 { character locators in the postamble }
define *char_loc0* = 246 { character locators in the postamble }
define *pre* = 247 { preamble }
define *post* = 248 { postamble beginning }
define *post-post* = 249 { postamble ending }
define *undefined-commands* ≡ 250, 251, 252, 253, 254, 255

18. 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$[\geq 4]$

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, which are exactly the values put into the header of any TFM file that shares information with this GF file. 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 $\leq$ -50. If there are no characters in the file, it's possible to have *min-m* > *max_m* and/ or min-n > max-n.)

19. 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^{24} 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.

20. 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. GFtoPK first reads the postamble, and then scans the file from front to back.

21. **Packed file format.** The packed file format is a compact representation of the data contained in a GF file. The information content is the same, but packed (PK) files are almost always less than half the size of their GF counterparts. They are also easier to convert into a raster representation because they do not have a profusion of *paint. skip,* and *new-row* commands to be separately interpreted. In addition, the PK format expressedly forbids special commands within a character. The minimum bounding box for each character is explicit in the format, and does not need to be scanned for as in the GF format. Finally, the width and escapement values are combined with the raster information into character "packets". making it simpler in many cases to process a character.

A PK file is organized as a stream of 8-bit bytes. At times, these bytes might be split into 4-bit nybbles or single bits, or combined into multiple byte parameters. When bytes are split into smaller pieces, the 'first' piece is always the most significant of the byte. For instance, the first bit of a byte is the bit with value 128; the first nybble can be found by dividing a byte by 16. Similarly, when bytes are combined into multiple byte parameters, the first byte is the most significant of the parameter. If the parameter is signed, it is represented by two's-complement notation.

The set of possible eight-bit values is separated into two sets, those that introduce a character definition, and those that do not. The values that introduce a character definition range from 0 to 239; byte values above 239 are interpreted as commands. Bytes that introduce character definitions are called flag bytes, and various fields within the byte indicate various things about how the character definition is encoded. Command bytes have zero or more parameters, and can never appear within a character definition or between parameters of another command, where they would be interpeted as data.

A PK file consists of a preamble, followed by a sequence of one or more character definitions, followed by a postamble. The preamble command must be the first byte in the file, followed immediately by its parameters. Any number of character definitions may follow, and any command but the preamble command and the postamble command may occur between character definitions. The very last command in the file must be the postamble.

22. The packed file format is intended to be easy to read and interpret by device drivers. The small size of the file reduces the input/ output overhead each time a font is loaded. For those drivers that load and save each font file into memory, the small size also helps reduce the memory requirements. The length of each character packet is specified, allowing the character raster data to be loaded into memory by simply counting bytes, rather than interpreting each command; then, each character can be interpreted on a demand basis. This also makes it possible for a driver to skip a particular character quickly if it knows that the character is unused.

23. First, the command bytes will be presented: then the format of the character definitions will be defined. Eight of the possible sixteen commands (values 240 through 255) are currently defined; the others are reserved for future extensions. The commands are listed below. Each command is specified by its symbolic name *(e.g., pk-no-op),* its opcode byte, and any parameters. The parameters are followed by a bracketed number telling how many bytes they occupy, with the number preceded by a plus sign if it is a signed quantity. (Four byte quantities are always signed, however.)

pk-xxx1 240 $k[1]$ $x[k]$. This command is undefined in general; it functions as a $(k + 2)$-byte no-op unless special PK-reading programs are being used. METAFONT generates xxx commands when encountering a special string. It is recommended that x be a string having the form of a keyword followed by possible parameters relevant to that keyword.

pk_xxx2 241 $k[2]$ $x[k]$. Like *pk-xxx1* , but $0 \le k < 65536$.

pk-xxx3 242 $k[3]$ $x[k]$. Like *pk_xxx1*, but $0 \le k < 2^{24}$. METRFONT uses this when sending a special string whose length exceeds 255.

pk_xxx4 243 $k[4]$ $x[k]$. Like *pk_xxx1*, but k can be ridiculously large; k musn't be negative.

pk-yyy 244 $y[4]$. This command is undefined in general; it functions as a five-byte *no-op* unless special PK 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.

pk-post 245. Beginning of the postamble. This command is followed by enough *pk-no-op* commands to make the file a multiple of four bytes long. Zero through three bytes are usual, but any number is allowed. This should make the file easy to read on machines that pack four bytes to a word.

pk-no-op 246. No operation, do nothing. Any number of *pk_no_op*'s may appear between PK commands, but a *pk-no-op* cannot be inserted between a command and its parameters, between two parameters, or inside a character definition.

pk-pre 247 $i[1]$ $k[1]$ $x[k]$ $ds[4]$ $cs[4]$ $hppp[4]$ $vppp[4]$. Preamble command. Here, i is the identification byte of the file, currently equal to 89. The string x is merely a comment, usually indicating the source of the PK file. The parameters ds and cs are the design size of the file in $1/2^{20}$ points, and the checksum of the file, respectively. The checksum should match the TFM file and the GF files for this font. Parameters *hppp* and *vppp* are the ratios of pixels per point, horizontally and vertically, multiplied by 2^{16}; they can be used to correlate the font with specific device resolutions, magnifications, and "at sizes". Usually. the name of the PK file is formed by concatenating the font name (e.g., amr10) with the resolution at which the font is prepared in pixels per inch multiplied by the magnification factor. and the letters PK. For instance, amr10 at 300 dots per inch should be named AMR10.300PK; at one thousand dots per inch and magstephalf, it should be named AMR10.1095PK.

24. We put a few of the above opcodes into definitions for symbolic use by this program.

define *pk-id* = 89 {the version of PK file described}
define *pk_xxx1* = 240 { special commands }
define *pk-yyy* = 244 {numspecial commands}
define *pk-post* = 245 {postamble}
define *pk-no-op* = 246 { no operation }
define *pk-pre* = 247 { preamble }

25. The PK format has two conflicting goals: to pack character raster and size information as compactly as possible, while retaining ease of translation into raster and other forms. A suitable compromise was found in the use of run-encoding of the raster information. Instead of packing the individual bits of the character. we instead count the number of consecutive 'black' or 'white' pixels in a horizontal raster row, and then encode this number. Run counts are found for each row from left to right, traversing rows from the top to bottom. This is the same way the GF format works. Instead of presenting each row individually, however, we concatenate all of the horizontal raster rows into one long string of pixels, and encode this row. With knowledge of the width of the bit-map, the original character glyph can be easily reconstructed. In addition, we do not need special commands to mark the end of one row and the beginning of the next.

Next, we place the burden of finding the minimum bounding box on the part of the font generator, since the characters will usually be used much more often than they are generated. The minimum bounding box is the smallest rectangle that encloses all 'black' pixels of a character. We also eliminate the need for a special end of character marker, by supplying exactly as many bits as are required to fill the minimum bounding box, from which the end of the character is implicit.

Let us next consider the distribution of the run counts. Analysis of several dozen pixel files at 300 dots per inch yields a distribution peaking at four, falling off slowly until ten, then a bit more steeply until twenty, and then asymptotically approaching the horizontal. Thus, the great majority of our run counts will fit in a four-bit nybble. The eight-bit byte is attractive for our run-counts, as it is the standard on many systems; however, the wasted four bits in the majority of cases seem a high price to pay. Another possibility is to use a Huffman-type encoding scheme with a variable number of bits for each run-count; this was rejected because of the overhead in fetching and examining individual bits in the file. Thus, the character raster definitions in the PK file format are based on the four-bit nybble.

26. An analysis of typical pixel files yielded another interesting statistic: Fully 37% of the raster rows were duplicates of the previous row. Thus, the PK format allows the specification of repeat counts, which indicate how many times a horizontal raster row is to be repeated. These repeated rows are taken out of the character glyph before individual rows are concatenated into the long string of pixels.

For elegance, we disallow a run count of zero. The case of a null raster description should be gleaned from the character width and height being equal to zero, and no raster data should be read. No other zero counts are ever necessary. Also, in the absence of repeat counts, the repeat value is set to be zero (only the original row is sent.) If a repeat count is seen, it takes effect on the current row. The current row is defined as the row on which the first pixel of the next run count will lie. The repeat count is set back to zero when the last pixel in the current row is seen, and the row is sent out.

This poses a problem for entirely black and entirely white rows, however. Let us say that the current row ends with four white pixels, and then we have five entirely empty rows, followed by a black pixel at the beginning of the next row, and the character width is ten pixels. We would like to use a repeat count, but there is no legal place to put it. If we put it before the white run count, it will apply to the current row. If we put it after, it applies to the row with the black pixel at the beginning. Thus, entirely white or entirely black repeated rows are always packed as large run counts (in this case, a white run count of 54) rather than repeat counts.

27. Now we turn our attention to the actual packing of the run counts and repeat counts into nybbles. There are only sixteen possible nybble values. We need to indicate run counts and repeat counts. Since the run counts are much more common, we will devote the majority of the nybble values to them. We therefore indicate a repeat count by a nybble of 14 followed by a packed number, where a packed number will be explained later. Since the repeat count value of one is so common, we indicate a repeat one command by a single nybble of 15. A 14 followed by the packed number 1 is still legal for a repeat one count. The run counts are coded directly as packed numbers.

For packed numbers, therefore, we have the nybble values 0 through 13. We need to represent the positive integers up to, say, $2^{31} - 1$. We would like the more common smaller numbers to take only one or two nybbles, and the infrequent large numbers to take three or more. We could therefore allocate one nybble value to indicate a large run count taking three or more nybbles. We do this with the value 0.

28. We are left with the values 1 through 13. We can allocate some of these, say *dyn-f,* to be one-nybble run counts. These will work for the run counts 1 . . *dyn-f.* For subsequent run counts, we will use a nybble greater than *dyn-f,* followed by a second nybble, whose value can run from 0 through 15. Thus, the two-nybble values will run from *dyn-f* + 1 . . (13 − *dyn-f*) ∗ 16 + *dyn-f.* We have our definition of large run count values now, being all counts greater than (13 − *dyn-f*) * 16 + *dyn-f.*

We can analyze our several dozen pixel files and determine an optimal value of *dyn-f*, and use this value for all of the characters. Unfortunately, values of *dyn-f* that pack small characters well tend to pack the large characters poorly, and values that pack large characters well are not efficient for the smaller characters. Thus, we choose the optimal *dyn-f* on a character basis, picking the value that will pack each individual character in the smallest number of nybbles. Legal values of *dyn-f* run from 0 (with no one-nybble run counts) to 13 (with no two-nybble run counts).

29. Our only remaining task in the coding of packed numbers is the large run counts. We use a scheme suggested by D. E. Knuth that simply and elegantly represents arbitrarily large values. The general scheme to represent an integer i is to write its hexadecimal representation, with leading zeros removed. Then we count the number of digits, and prepend one less than that many zeros before the hexadecimal representation. Thus, the values from one to fifteen occupy one nybble; the values sixteen through 255 occupy three, the values 256 through 4095 require five, etc.

For our purposes, however, we have already represented the numbers one through (13 − *dyn-f*) ∗ 16 + *dyn-f*. In addition, the one-nybble values have already been taken by our other commands, which means that only the values from sixteen up are available to us for long run counts. Thus, we simply normalize our long run counts, by subtracting (13 − *dyn-f*) ∗ 16 + *dyn-f* + 1 and adding 16, and then we represent the result according to the scheme above.

30. The final algorithm for decoding the run counts based on the above scheme might look like this, assuming that a procedure called *pk-nyb* is available to get the next nybble from the file, and assuming that the global *repeat-count* indicates whether a row needs to be repeated. Note that this routine is recursive, but since a repeat count can never directly follow another repeat count, it can only be recursive to one level.

```
@{
function pk-packedsum: integer;
  var i, j, k: integer;
  begin i + get-nyb;
  if $i = 0$ then
    begin repeat j + get-nyb; incr(i);
    until $j \neq 0$;
    while $i > 0$ do
      begin $j \leftarrow j * 16$ + get-nyb; decr(i);
      end;
    pk-packed-num $\leftarrow j - 15 + (13 - dyn\text{-}f) * 16 + dyn\text{-}f$;
    end
  else if $i \leq dyn\text{-}f$ then pk-packed-num $\leftarrow i$
    else if $i < 14$ then pk-packed-num $\leftarrow (i - dyn\text{-}f - 1) * 16 + get\text{-}nyb + dyn\text{-}f + 1$
      else begin if $i = 14$ then repeat-count $\leftarrow$ pk-packed-num
        else repeat-count $\leftarrow 1$;
        pk-packed-num $\leftarrow$ pk-packed-num:
        end;
  end:
@}
```

31. For low resolution fonts, or characters with 'gray' areas, run encoding can often make the character many times larger. Therefore, for those characters that cannot be encoded efficiently with run counts, the PK format allows bit-mapping of the characters. This is indicated by a *dyn-f* value of 14. The bits are packed tightly, by concatenating all of the horizontal raster rows into one long string, and then packing this string eight bits to a byte. The number of bytes required can be calculated by $(width * height + 7)$ div 8. This format should only be used when packing the character by run counts takes more bytes than this, although, of course. it is legal for any character. Any extra bits in the last byte should be set to zero.

32. At this point, we are ready to introduce the format for a character descriptor. It consists of three parts: a flag byte, a character preamble, and the raster data. The most significant four bits of the flag byte yield the *dyn-f* value for that character. (Notice that only values of 0 through 14 are legal for *dyn-f,* with 14 indicating a bit mapped character; thus, the flag bytes do not conflict with the command bytes, whose upper nybble is always 15.) The next bit (with weight 16) indicates whether the first run count is a black count or a white count, with a one indicating a black count. For bit-mapped characters, this bit should be set to a zero. The next bit (with weight 8) indicates whether certain later parameters (referred to as size parameters) are given in one-byte or two-byte quantities, with a one indicating that they are in two-byte quantities. The last two bits are concatenated on to the beginning of the length parameter in the character preamble, which will be explained below.

However, if the last three bits of the flag byte are all set (normally indicating that the size parameters are two-byte values and that a 3 should be prepended to the length parameter), then a long format of the character preamble should be used instead of one of the short forms.

Therefore, there are three formats for the character preamble; the one that is used depends on the least significant three bits of the flag byte. If the least significant three bits are in the range zero through three, the short format is used. If they are in the range four through six, the extended short format is used. Otherwise, if the least significant bits are all set, then the long form of the character preamble is used. The preamble formats are explained below.

Short form: ***flag***[1] ***pl***[1] *cc*[1] ***tfm***[3] *dm*[1] *w*[1] *h*[1] ***hoff***[+1] ***voff***[+1]. If this format of the character preamble is used, the above parameters must all fit in the indicated number of bytes, signed or unsigned as indicated. Almost all of the standard TEX font characters fit; the few exceptions are fonts such as `aminch`.

Extended short form: ***flag***[1] ***pl***[2] *cc*[1] ***tfm***[3] *dm*[2] *w*[2] *h*[2] ***hoff***[+2] ***voff***[+2]. Larger characters use this extended format.

Long form: ***flag***[1] ***pl***[4] *cc*[4] ***tfm***[4] *dx*[4] *dy*[4] *w*[4] *h*[4] ***hoff***[4] ***voff***[4]. This is the general format that allows all of the parameters of the GF file format, including vertical escapement.

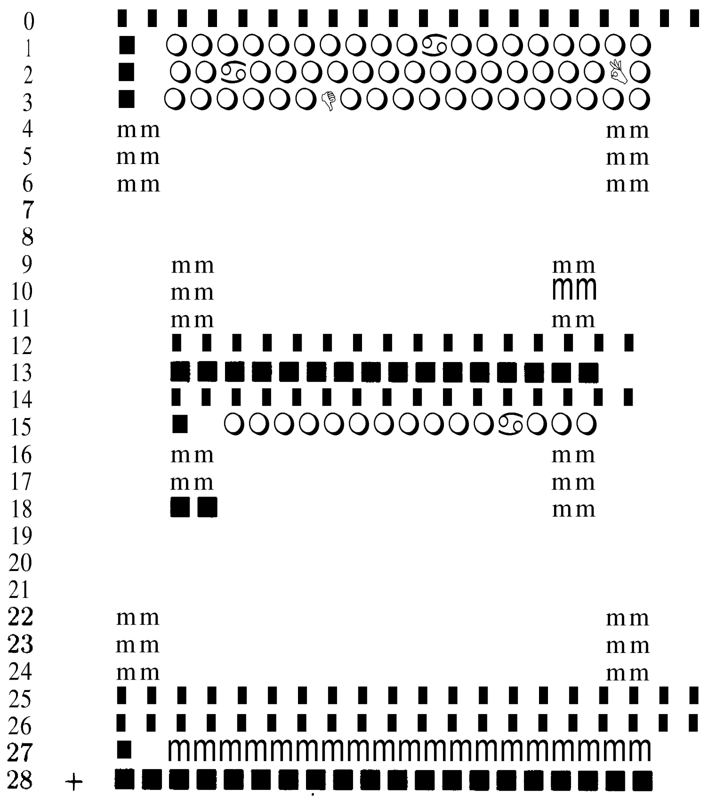
The ***flag*** parameter is the flag byte. The parameter ***pl*** (packet length) contains the offset of the byte following this character descriptor, with respect to the beginning of the *tfm* width parameter. This is given so a PK reading program can, once it has read the flag byte, packet length, and character code (cc), skip over the character by simply reading this many more bytes. For the two short forms of the character preamble, the last two bits of the flag byte should be considered the two most-significant bits of the packet length. For the short format, the true packet length might be calculated as (***flag*** mod 4) $*$ 256 + ***pl***; for the short extended format, it might be calculated as (***flag*** mod 4) $*$ 65536 + ***pl***.

The w parameter is the width and the *h* parameter is the height in pixels of the minimum bounding box. The *dx* and *dy* parameters are the horizontal and vertical escapements, respectively. In the short formats, *dy* is assumed to be zero and *dm* is *dx* but in pixels; in the long format, *dx* and *dy* are both in pixels multiplied by 2^{16}. The ***hoff*** is the horizontal offset from the upper left pixel to the reference pixel; the ***voff*** is the vertical offset. They are both given in pixels, with right and down being positive. The reference pixel is the pixel that occupies the unit square in METAFONT; the METRFONT reference point is the lower left hand corner of this pixel. (See the example below.)

33. TEX requires all characters that have the same character codes modulo 256 to have also the same *tfm* widths and escapement values. The PK format does not itself make this a requirement, but in order for the font to work correctly with the TEX software, this constraint should be observed. (The standard version of TEX cannot output character codes greater than 255, but extended versions do exist.)

Following the character preamble is the raster information for the character, packed by run counts or by bits, as indicated by the flag byte. If the character is packed by run counts and the required number of nybbles is odd, then the last byte of the raster description should have a zero for its least significant nybble.

34. As an illustration of the PK format, the character Ξ from the font amr10 at 300 dots per inch will be encoded. This character was chosen because it illustrates some of the borderline cases. The raster for the character looks like this (the row numbers are chosen for convenience, and are not METAFONT's row numbers.)



The width of the minimum bounding box for this character is 20; its height is 29. The '+' represents the reference pixel; notice how it lies outside the minimum bounding box. The *hoff* value is -2, and the *voff* is 28.

The first task is to calculate the run counts and repeat counts. The repeat counts are placed at the first transition (black to white or white to black) in a row, and are enclosed in brackets. White counts are enclosed in parentheses. It is relatively easy to generate the counts list:

82 [2] (16) 2 (42) [2] 2 (12) 2 (4) [3]
16 (4) [2] 2 (12) 2 (62) [2] 2 (16) 82

Note that any duplicated rows that are not all white or all black are removed before the repeat counts arc calculated. The rows thus removed are rows 5, 6, 10, 11, 13, 14, 15, 17, 18, 23, and 24.

35. The next step in the encoding of this character is to calculate the optimal value of *dyn-f*. The details of how this calculation is done are not important here: suffice it to say that there is a simple algorithm that can determine the best value of *dyn-f* in one pass over the count list. For this character. the optimal value turns out to be 8 (atypically low). Thus, all count values less than or equal to 8 are packed in one nybble; those from nine to $(13 - 8) * 16 + 8$ or 88 are packed in two nybbles. The run encoded values now become (in hex, separated according to the above list):

```
D9 E2 97 2 B1 E2 2 93 2 4 E3
97 4 E2 2 93 2 C5 E2 2 97 D9
```

which comes to 36 nybbles, or 18 bytes. This is shorter than the 73 bytes required for the bit map, so we use the run count packing.

36. The short form of the character preamble is used because all of the parameters fit in their respective lengths. The packet length is therefore 18 bytes for the raster, plus eight bytes for the character preamble parameters following the character code, or 26. The *tfm* width for this character is 640796, or 9C71C in hexadecimal. The horizontal escapement is 25 pixels. The flag byte is 88 hex, indicating the short preamble, the black first count, and the *dyn-f* value of 8. The final total character packet, in hexadecimal, is:

Flag byte	88
Packet length	1A
Character code	04
tfm width	09 c7 1C
Horizontal escapement (pixels)	19
Width of bit map	14
Height of bit map	1D
Horizontal offset (signed)	FE
Vertical offset	1C
Raster data	D9 E2 97
	2B 1E 22
	93 24 E3
	97 4E 22
	93 2C 5E
	22 97 D9

37. Input and output for binary files. We have seen that a GF file is a sequence of 8-bit bytes. The bytes appear physically in what is called a 'packed file of 0 . . 255' in Pascal lingo. The PK file is also a sequence of 8-bit bytes.

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 systern-dependent code is often needed to deal with binary files, even though most of the program in this section of GFtoPK is written in standard Pascal.

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}

38. The program deals with two binary file variables: *gf_file* is the input file that we are translating into PK format, to be written on *pk_file* .

⟨ Globals in the outer block 11 ⟩ +≡
gf-file: *byte_file*; { the stuff we are GFtoPKing }
pk-file: byte-file; { the stuff we have GFtoPKed }

39. To prepare the *gf-file* for input, we *reset* it.

procedure *open_gf_file* : { prepares to read packed bytes in *gf-file* }
  begin *reset*(*gf_file*); *gf-Zoc* ← 0;
  end:

40. To prepare the *pk-file* for output, we *rewrite* it.

procedure *open_pk_file*; { prepares to write packed bytes in *pk_file* }
  begin *rewrite*(*pk_file*); *pk_loc* ← *0; pk-open* ← *true;*
  end;

41. The variable *pk-Zoc* contains the number of the byte about to be written to the *pk-file,* and *gf-Zoc* is the byte about to be read from the *gf_file*. Also, *pk-open* indicates that the packed file has been opened and is ready for output.

⟨ Globals in the outer block 11 ⟩ +≡
pk-Zoc: *integer;* { where we are about to write, in *pk_file* }
gf-Zoc: *integer;* { where are we in the *gf_file*}
pk_open: *boolean;* { is the packed file open? }

42. We do not open the *pk-file* until after the postarnble of the *gf_file* has been read. This can be used, for instance, to calculate a resolution to put in the *suffix* of the *pk-file* name. This also means, however, that specials in the postamble (which METAFONT never generates) do not get sent to the *pk-file.*

⟨ Set initial values 12 ⟩ +≡
  pk-open ← *false;*

43. We shall use two simple functions to read the next byte or bytes from *gf_file*. We either need to get an individual byte or a set of four bytes.

```
function gf_byte: integer;  { returns the next byte, unsigned }
  var b: eight_bits;
  begin if eof(gf_file) then bad_gf('Unexpected␣end␣of␣file!')
  else begin read(gf_file, b); gf_byte ← b;
    end;
  incr(gf_loc);
  end;

function gf_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);
  if a < 128 then gf_signed_quad ← ((a * 256 + b) * 256 + c) * 256 + d
  else gf_signed_quad ← (((a − 256) * 256 + b) * 256 + c) * 256 + d;
  gf_loc ← gf_loc + 4;
  end;
```

44. We also need a few routines to write data to the PK file. We write data in 4-, 8-, 16-, 24-. and 32-bit chunks, so we define the appropriate-routines. We must be careful not to let the sign bit mess us up, as some Pascals implement division of a negative integer differently.

```
procedure pk_byte (a : integer);
  begin if pk-open then
    begin if a < 0 then a + a + 256;
    write (pk-file, a); incr(pk_loc);
    end;
  end;
procedure pk_halfword (a : integer);
  begin if a < 0 then a ← a + 65536;
  write (pk-file , a div 256); write (pk-file, a mod 256); pk-lot ← pk-lot + 2;
  end;
procedure pk_three_bytes(a : integer);
  begin write (pk-file, a div 65536 mod 256); write (pk-file, a div 256 mod 256); write(pk_file, a mod 256):
  pk_loc + pk_loc + 3;
  end;
procedure pk_word (a : integer);
  var b: integer;
  begin if pk-open then
    begin if a < 0 then
      begin a ← a + '10000000000 ; a ← a + '10000000000; b + 128 + a div 16777216;
      end
    else b ← a div 16777216;
    write (pk-file, b); write (pk-file, a div 65536 mod 256): write (pk-file, a div 256 mod 256);
    write(pk_file, a mod 256); pk_loc ← pk loc + 4;
    end;
  end:
procedure pk-nyb(u : integer);
  begin if bit-weight = 16 then
    begin output-byte ← a * 16; bit-weight ← 1;
    end
  else begin pk_byte( output-byte + a); bit-weight ← 16;
    end;
  end;
```

45. We need the globals *bit-weight* and *output-byte* for buffering.

```
(Globals in the outer block 11) +≡
bit-weight: integer; { output bit weight }
output-byte: integer; { output byte for pk file }
```

46. Finally we come to the routines that are used for random access of the *gf_file*. To correctly find and read the postamble of the file, we need two routines, one to find the length of the *gf_file*, and one to position the *gf_file*. We assume that the first byte of the file is numbered zero.

Such routines are, of course, highly system dependent. They are implemented here in terms of two assumed system routines called *set-pos* and *cur-pos*. The call *set_pos*(f, n) moves to item n in file f, unless n is negative or larger than the total number of items in f; in the latter case, *set_pos*(f, n) moves to the end of file f. The call *cur_pos*(f) gives the total number of items in f, if *eof* (f) is true; we use *cur-pos* only in such a situation.

```
procedure find_gf_length;
  begin set-pos (gf_file , − 1); gf_len ← cur-pas (gf-file );
  end;

procedure move-to-byte(n : integer);
  begin set-pos (gf-file , n); gf_loc ← n;
  end;
```

47. The global *gf_len* contains the final total length of the *gf-file*.

```
⟨ Globals in the outer block 11 ⟩ +≡
gf_len: integer;   { length of gf-file }
```

48. Plan of attack. It would seem at first that converting a GF file to PK format should be relatively easy, since they both use a form of run-encoding. Unfortunately, several idiosyncracies of the GF format make this conversion slightly cumbersome. The GF format separates the raster information from the escapement values and TFM widths: the PK format combines all information about a single character into one character packet. The GF run-encoding is on a row-by-row basis, and the PK format is on a glyph basis, as if all of the raster rows in the glyph were concatenated into one long row. The encoding of the run-counts in the GF files is fixed, whereas the PK format uses a dynamic encoding scheme that must be adjusted for each character. And, finally, any repeated rows can be marked and sent with a single command in the PK format.

There are four major steps in the conversion process. First, the postamble of the *gf-file* is found and read, and the data from the character locators is stored in memory. Next, the preamble of the *pk-file* is written. The third and by far the most difficult step reads the raster representation of all of the characters from the GF file, packs them, and writes them to the *pk-file*. Finally, the postamble is written to the *pk-file*.

The conversion of the character raster information from the *gf-file* to the format required by the *pk-file* takes several smaller steps. The GF file is read, the commands are interpreted, and the run counts are stored in the working *row* array. Each row is terminated by a *end-of-row* value, and the character glyph is terminated by an *end-of-char* value. Then, this representation of the character glyph is scanned to determine the minimum bounding box in which it will fit, correcting the *min_m*, *max_m*, *min_n*, and *max_n* values, and calculating the offset values. The third sub-step is to restructure the row list from a list based on rows to a list based on the entire glyph. Then, an optimal value of ***dyn_f*** is calculated, and the final size of the counts is found for the PK file format, and compared with the bit-wise packed glyph. If the run-encoding scheme is shorter, the character is written to the *pk-file* as row counts; otherwise, it is written using a bit-packed scheme.

To save various information while the GF file is being loaded, we need several arrays. The *tfm-width*, *dx*, and *dy* arrays store the obvious values. The *status* array contains the current status of the particular character. A value of 0 indicates that the character has never been defined; a 1 indicates that the character locator for that character was read in; and a 2 indicates that the raster information for at least one character was read from the *gf_file* and written to the *pk-file*. The *row* array contains row counts. It is filled anew for each character, and is used as a general workspace. The GF counts are stored starting at location 2 in this array, so that the PK counts can be written to the same array, overwriting the GF counts, without destroying any counts before they are used. (A possible repeat count in the first row might make the first row of the PK file one count longer; all succeeding rows are guaranteed to be the same length or shorter because of the *end-of-row* flags in the GF format that are unnecessary in the PK format.)

```
  define virgin ≡ 0   { never heard of this character yet }
  define located ≡ 1   { locators read for this character}
  define sent ≡ 2   { at least one of these characters has been sent }
⟨ Globals in the outer block 11 ⟩ +≡
tfm-width: array [0 , . 255] of integer;   { the TFM widths of characters }
dx, dy: array [0 . . 255] of integer;   { the horizontal and vertical escapements }
status: array [0 . . 255] of virgin . . sent;   { character status }
row: array [0 . . mux-row] of integer;   { the row counts for working }
```

49. Here we initialize all of the character *status* values to *virgin*.

```
⟨ Set initial values 12 ⟩ +≡
  for i ← 0 to 255 do status[i] ← virgin:
```

50. And, finally, *we* need to define the *end_of_row* and *end-of-char* values. These cannot be values that can be taken on either by legitimate run counts, even when wrapping around an entire character. Nor can they be values that repeat counts can take on. Since repeat counts can be arbitrarily large, we restrict ourselves to negative values whose absolute values are greater than the largest possible repeat count.

```
  define end-of-row ≡ (-99999)   { indicates the end of a row }
  define end-of-char ≡ (-99998)   { indicates the end of a character }
```

51. Reading the generic font file. There are two major procedures in this program that do all of the work. The first is *convert-gf-file,* which interprets the GF commands and puts row counts into the *row* array. The second. which we only anticipate at the moment, actually packs the row counts into nybbles and writes them to the packed file.

```
  ( Packing procedures 62);
procedure convert-gf-file ;
  var i, j, k: integer;  { general purpose indices }
    gf-corn: integer;  {current gf command }
    ( Locals to convert-gf-file 58)
    begin open-gf-file ;
    if gf-byte ≠ pre then bad-gf ('First␣byte␣is␣not␣preamble');
    if gf-byte ≠ gf_id_byte then bad-gf ('Identification␣byte␣is␣incorrect ');
    ( Find and interpret postamble 60);
    open-pk-file; ( Write preamble 81);
    move-to-byte(2); i ← gf-byte; print( ´ ´ ´ ´);
    for j ← 1 to i do print (xchr[gf_byte]);
    print-Zn( ´ ´ ´ ´);
    repeat gf-corn + gf-byte;
      case gf-corn of
      boc, boc1 : ( Interpret character 54 );
        ( Specials and no-op cases 53 );
      post: ; {we will actually do the work for this one later }
      othercases bad-gf ( 'Unexpected,', gf-corn : 1, '␣command␣between␣characters')
      endcases;
    until gf-corn = post;
    (Write postamble 84);
    end;
```

52. We need a few easy macros to expand some case statements:

```
  define four-cases(#) ≡ #, # + 1, # + 2, # + 3
  define sixteen-cases (#) ≡ four-cases (#), four-cases (# + 4), four-cases (# + 8), four-cases (# + 12)
  define sixty-four-cases (#) ≡ sixteen-cases (#), sixteen-cases (# + 16), sixteen-cases (# + 32),
            sixteen-cases (# + 48)
  define one-sixty&e-cases(#) ≡ sixty-four-cases(#), sixty-four-cases(# + 64), sixteen_cases( # + 128),
            sixteen-cases (# + 144), four_cases(# + 160), # + 164
```

53. In this program, all special commands are passed unchanged and any *no-op* bytes are ignored, so we write some code to handle these:

```
( Specials and no_op cases 53) ≡
four_cases(xxx1 ): begin pk_byte(gf_com − xxx1 + pk_xxx1 ); i + 0;
  for j ← 0 to gf-corn − xxx1 do
    begin k + gf-byte; pk_byte(k); i ← i * 256 + k;
    end;
  for j + 1 to i do pk_byte(gf_byte);
  end;
yyy: begin pk-byte(pk-yyy); pk_word(gf_signed_quad);
  end;
no-op:
```

This code is used in sections 51, 54, 57, and 60.

54. Now we need the routine that handles the character commands. Again, only a subset of the gf commands are permissible inside character definitions, so we only look for these.

```
⟨ Interpret character 54⟩ ≡
  begin if gf_com = boc then
    begin gf-ch ← gf-signed-quad; i ← gf-signed-quad; { dispose of back pointer }
    min-m + gf-signed-quad; max-m + gf-signed-quad ; min-n + gf_signed_quad;
    max-n + gf-signed-quad;
    end
  else begin gf-ch ← gf-byte; i + gf-byte; max-m ← gf-byte; min-m + max-m − i; i + gf-byte:
    max-n ← gf-byte; min-n + max-n − i;
    end;
  d_print_ln( ´Character␣ ´, gf-ch : 1); gf-ch-mod-256 + gf-ch mod 256;
  if status [gf_ch_mod_256] = virgin then
    begin bad-gf ( ´no␣character␣locator␣for␣character␣´, gf-ch : 1);
    repeat gf-corn + gf-b yte;
      case gf-corn of
      sixty-four-cases(paint-0), eoc, skip0, one-sixty-five-cases( new_row_0): ;
        ⟨ Specials and no-op cases 53 ⟩;
      paintl, skip1 : i + gf-byte;
      paint1 + 1, skip1 + 1: begin i ← gf-byte; i ← gf-byte;
        end;
      paint1 + 2, skip1 + 2: begin i ← gf-byte; i + gf-byte; i ← gf-byte;
        end;
      othercases bad-gf (´Unexpected,´, gf-corn : 1, ´␣while␣skipping␣character´)
      endcases;
    until gf-corn = eoc
    end
  else ⟨ Convert character to packed form 57⟩;
  end
```

This code is used in section 51.

55. Communication between the procedures *convert-gf-file* and *pack-and-send-character* is done with a few global variables.

```
⟨ Globals in the outer block 11⟩ +≡
gf_ch : integer;  { the character we are working with }
gf_ch_mod_256: integer;  { locater pointer }
pred_pk_loc: integer;  { where we predict the end of the character to be. }
max-n, min-n: integer;  { the maximum and minimum horizontal rows }
max-m, min-m : integer;  { the maximum and minimum vertical rows }
row-ptr: integer;  { where we are in the row array. }
```

56. Now we are at the beginning of a character that we need the raster for. Before we get into the complexities of decoding the *paint, skip,* and *new-row* commands, let's define a macro that will help us fill up the *row* array. Note that we check that *row-ptr* never exceeds *max-row;* Instead of calling *bad-gf* directly, as this macro is repeated eight times, we simply set the *bad* flag true.

```
define put-in-rows (#) ≡
        begin if row-ptr > max-row then bad + true
        else begin row [row-ptr] ← #; incr (row-ptr);
          end;
        end
```

57. Now we have the procedure that decodes the various commands and puts counts into the *row* array. This would be a trivial procedure, except for the *paint-0* command. Because the *paint-O* command exists, it is possible to have a sequence like *paint 42. paint-o, paint 38, paint-O,* ***paint_0***, ***paint_0***, *paint 33, skip-o.* This would be an entirely empty row, but if we left the zeros in the row array, it would be difficult to recognize the row as empty.

This type of situation probably would never occur in practice, but it is defined by the GF format, so we must be able to handle it. The extra code is really quite simple, just difficult to understand; and it does not cut down the speed appreciably. Our goal is this: to collapse sequences like *paint 42,* ***paint_0***. *paint* 32 to a single count of 74, and to insure that the last count of a row is a black count rather than a white count. A buffer variable *extra,* and two state flags, *on* and *state,* enable us to accomplish this.

The *on* variable is essentially the *paint-switch* described in the GF description. If it is true, then we are currently painting black pixels. The *extra* variable holds a count that is about to be placed into the row array. We hold it in this array until *we* get a *paint* command of the opposite color that is greater than 0. If *we* get a *paint-O* command, then the *state* flag is turned on, indicating that the next count we receive can be added to the *extra* variable as it is the same color.

```
⟨ Convert character to packed form 57⟩ ≡
 begin bad ← false; row-ptr ← 2; on ← false; extra ← 0; state ← true;
 repeat gf-corn ← gf-byte;
   case gf-corn of
   ⟨ Cases for paint commands 59⟩;
   four_cases(skip0): begin i ← 0;
     for j ← 1 to gf-corn − skip0 do i ← i * 256 + gf-byte;
     if ¬on A (extra > 0) then put_in_rows(extra);
     for j ← 0 to i do put_in_rows(end-of-row);
     on ← false; extra ← 0; state ← true;
     end;
   one-sixty-five-cases(new_row_0): begin if ¬on A (extra > 0) then put-in-rows (extra);
     put-in-rows(end-of-row); on ← true; extra ← gf-corn − new-row-O; state ← false;
     end;
   ⟨ Specials and no-op cases 53⟩;
   eoc: begin if ¬on A (extra > 0) then put_in_rows(extra);
     if (row-ptr > 2) A (row[row_ptr − 1] ≠ end-of-row) then put_in_rows(end-of-row);
     put-in-rows (end-of-char);
     if bad then abort('Ran␣out␣of␣internal␣memory␣f or␣row␣counts !');
     pack-and-send-character; status [gf_ch_mod_256] ← sent;
     if pk_loc ≠ pred_pk_loc then abort('Internal␣error␣while␣writing␣character!');
     end:
   othercases bad-gf ('Unexpected␣', gf-corn : 1, '␣command␣in␣character␣definition')
   endcases;
 until gf-corn = eoc;
 end
```

This code is used in section 54.

58. A few more locals used above and below:

```
⟨ Locals to convert_gf_file 58⟩ ≡
on: boolean; {indicates whether we are white or black}
state: boolean; { a state variable-is the next count the same race as the one in the extra buffer? }
extra: integer; {where we pool our counts}
bad: boolean; {did we run out of space?}
```

See also section 61.

This code is used in section 51.

59. ⟨ Cases for *paint* commands 59⟩ ≡

```
paint-0: begin state ← ¬state; on ← ¬on;
  end;
sixty-four-cases(paint-0 + 1), paint1 + 1, paint1 + 2: begin if gf-com < paint1 then i ← gf-com − paint-0
  else begin i ← 0;
    for j ← 0 to gf-com − paint1 do i ← i * 256 + gf-byte;
    end;
  if state then
    begin extra ← extra + i; state ← false;
    end
  else begin put-in-rows(extra); extra ← i;
    end;
  on ← ¬on;
  end
```

This code is used in section 57.

60. Our last remaining task is to interpret the postamble commands. The only things that may appear in the postamble are *post-post, char-Zoc. char_loc0*, and the special commands. Note that any special commands that might appear in the postamble are not written to the *pk_file*. Since METRFONT does not generate special commands in the postamble, this should not be a major difficulty.

⟨ Find and interpret postamble 60⟩ ≡

```
find-qf-lenqth; post-lot ← gf-Zen − 4;
repeat if post-Zoc = 0 then bad-gf('all␣223''s');
  move_to_byte(post_loc); k ← gf-byte; decr(post_loc);
until k ≠ 223;
if k ≠ gf_id_byte then bad-gf('ID␣byte␣is␣', k : 1);
move-to-byte (post-lot − 3); q + gf-signed-quad;
if (q < 0) V (q > post-Zoc − 3) then bad-gf('post␣pointer␣is␣', q : 1);
move-to-byte(q); k + gf-byte;
if k ≠ post then bad-gf('byte␣at␣', q : 1, '␣is␣not␣post');
i ← gf-signed-quad; { skip over junk }
design-size ← gf-signed-quad; check-sum ← gf-signed-quad; hppp ← gf-signed-quad;
h_mag ← round (hppp * 72.27/65536); vppp ← gf-signed-quad;
if hppp ≠ vppp then print_ln('Odd␣aspect␣ratio!');
i + gf-signed-quad; i ← gf-signed-quad; { skip over junk }
i ← gf-signed-quad; i ← gf-signed-quad;
repeat gf-corn ← gf-byte;
  case gf-corn of
  char-Zoc, char_loc0: begin gf-ch ← gf-byte;
    if status [gf-ch] ≠ virgin then bad-gf('Locator␣f or␣this␣character␣already␣f dund.');
    if gf-corn = char-Zoc then
      begin dx[gf_ch] ← gf-signed-quad; dy[gf_ch] ← gf-signed-quad;
      end
    else begin dx [gf-ch] + gf-byte * 65536; dy[gf_ch] ← 0;
      end;
    tfm-width [gf-ch] ← gf-signed-quad; i ← gf-signed-quad; status [gf-ch] ← located;
    end;
    ⟨ Specials and no-op cases 53⟩;
  post-post: ;
  othercases bad-gf ('Unexpected,', gf-corn : 1, '␣in␣postamble')
  endcases;
until gf-corn = post-post
```

This code is used in section 51.

61. Just a few more locals:

⟨ Locals to *convert_gf_file* 58⟩ +≡

```
hppp , vppp: integer; { horizontal and vertical pixels per point }
q: integer; { quad temporary }
post_loc: integer; { where the postamble was }
```

62. Converting the counts to packed format. This procedure is passed the set of row counts from the GF file. It writes the character to the PK file. First, the minimum bounding box is determined. Next, the row-oriented count list is converted to a count list based on the entire glyph. Finally, we calculate the optimal *dyn-f* and send the character.

```
⟨ Packing procedures 62⟩ ≡
procedure pack-and-send-character;
  var i, j, k: integer;  {general indices}
    ⟨ Locals to pack-and-send-character 65 ⟩
    begin ⟨ Scan for bounding box 63⟩;
    ⟨ Convert row-list to glyph-list 64⟩;
    ⟨ Calculate dyn-f and packed size and write character 68⟩;
    end
```

This code is used in section 51.

63. Now we have the row counts in our *row* array. To find the real *mux-n, we* look for the first non-*end-of-row* value in the *row.* If it is an *end-of-char,* the entire character is blank. Otherwise, we first eliminate all of the blank rows at the end of the character. Next, for each remaining row, we check the first white count for a new *min-m,* and the total length of the row for a new *max_m*.

```
⟨ Scan for bounding box 63⟩ ≡
  i ← 2; decr(row-ptr);
  while row [i] = end-of-row do incr(i);
  if row [i] ≠ end-of-char then
    begin mux-n ← max_n − i + 2;
    while row [row-ptr − 2] = end-of-row do
      begin decr( row-ptr); row [row-ptr] ← end-of-char;
      end;
    min-n ← mux-n + 1; extra ← mux-m − min-m + 1; mux-m ← 0; j ← i;
    while row [j] ≠ end-of-chur do
      begin decr (min-n);
      if row[j] ≠ end-of-row then
        begin k ← row[j];
        if k < extra then extra ← k;
        incr (j);
        while row [j] ≠ end-of-row do
          begin k ← k + row [j]; incr (j);
          end;
        if mux-m < k then mux-m ← k;
        end;
      incr (j);
      end:
    min-m ← min-m + extra; mux-m ← min-m + mux-m − 1 − extra; height ← max-n − min_n + 1;
    width ← mux-m − min-m + 1; x-offset ← −min_m; y-offset ← mux-n;
    d-print-ln( ´W␣´, width : 1, ´␣H␣´, height : 1, ´␣X␣´, x_offset : 1, ´␣Y␣´, y_offset : 1);
    end
  else begin height ← 0; width ← 0; x_offset ← 0; y_offset ← 0; d_print_ln( ´Empty␣raster.´);
    end
```

This code is used in section 62.

64. We must convert the run-count array from a row orientation to a glyph orientation, with repeat counts for repeated rows. We seperate this task into two smaller tasks, on a per row basis. But first, we define a new macro to help us fill up this new array. Here, we have no fear that we will run out of space, as the glyph representation is provably smaller than the rows representation.

```
define put-count (#) ≡
        begin row[put_ptr] ← #; incr(put_ptr);
        if repeat-flag > 0 then
          begin row [put-ptr] ← -repeat-flag; repent-flag ← 0; incr(put_ptr);
          end;
        end
```

```
⟨ Convert row-list to glyph-list 64⟩ ≡
  put-ptr ← 0; row-ptr ← 2; repeat-flag ← 0; state ← true; buff ← 0;
  while row [row_ptr] = end-of-row do incr( row-ptr);
  while row [row-ptr] ≠ end-of-char do
    begin ⟨ Skip over repeated rows 66⟩;
    ⟨ Reformat count list 67 ⟩;
    end;
  if buff > 0 then put-count( buff );
  put-count (end-of-char)
```

This code is used in section 62.

65. Some more locals for *pack-and-send-character* used above:

```
⟨ Locals to puck-and-send-character 65⟩ ≡
extra: integer; { little buffer for count values }
put-ptr: integer; { next location to fill in row }
repeat-flag: integer; { how many times the current row is repeated }
h-bit: integer; { horizontal bit count for each row }
buff: integer; { our count accumulator }
```

See also sections 70 and 77.

This code is used in section 62.

66. In this short section of code, we are at the beginning of a new row. We scan forward, looking for repeated rows. If there are any, *repeat_flag* gets the count, and the *row-ptr* points to the beginning of the last of the repeated rows. Two points must be made here. First, we do not count all-black or all-white rows as repeated, as a large "paint" count will take care of them, and also there is no black to white or white to black transition in the row where we could insert a repeat count. That is the meaning of the big if statement that conditions this section. Secondly, the while $row[i] = row[j]$ do loop is guaranteed to terminate, as $j > i$ and the character is terminated by a unique *end-of-char* value.

```
⟨ Skip over repeated rows 66⟩ ≡
  i ← row-ptr;
  if (row[i] ≠ end-of-row) A ((row [i] ≠ extra) V (row [i + 1] ≠ width)) then
    beginj ← i + 1;
    while row [j − 1] ≠ end-of-row do incr(j);
    while row [i] = row [j] do
      begin if row [i] = end-of-row then
        begin incr(repeat_flag); row_ptr ← i + 1;
        end;
      incr(i); incr(j);
      end;
    end
```

This code is used in section 64.

67. Here we actually spit out a row. The routine is somewhat similar to the routine where we actually interpret the GF commands in the count buffering. We must make sure to keep track of how many bits have actually been sent, so when we hit the end of a row, we can send a white count for the remaining bits, and possibly add the white count of the next row to it. And, finally, we must not forget to subtract the *extra* white space at the beginning of each row from the first white count.

```
⟨ Reformat count list 67⟩ ≡
  if $row[row\_ptr] \neq end\_of\_row$ then $row[row\_ptr] \leftarrow row[row\_ptr] - extra$;
  $h\_bit \leftarrow 0$;
  while $row[row\_ptr] \neq end\_of\_row$ do
    begin $h\_bit \leftarrow h\_bit + row[row\_ptr]$;
    if $state$ then
      begin $buff \leftarrow buff + row[row\_ptr]$; $state \leftarrow false$;
      end
    else if $row[row\_ptr] > 0$ then
        begin $put\_count(buff)$; $buff \leftarrow row[row\_ptr]$;
        end
      else $state \leftarrow true$;
    $incr(row\_ptr)$;
    end;
  if $h\_bit < width$ then
    if $state$ then $buff \leftarrow buff + width - h\_bit$
    else begin $put\_count(buff)$; $buff \leftarrow width - h\_bit$; $state \leftarrow true$;
      end
  else $state \leftarrow false$;
  $incr(row\_ptr)$
```

This code is used in section 64.

68. Here is another piece of rather intricate code. We determine the smallest size in which we can pack the data, calculating *dyn-f* in the process. To do this, we calculate the size required if *dyn-f* is 0. and put this in *camp-size.* Then, we calculate the changes in the size for each increment of *dyn-f* , and stick these values in the *deriv* array. Finally, we scan through this array and find the final minimum value, which we then use to send the character data.

```
⟨ Calculate dyn-f and packed size and write character 68 ⟩ ≡
  for $i \gets 1$ to 13 do $deriv[i] \gets 0$;
  $i \gets 0$; $first\text{-}on \gets row[i] = 0$;
  if first_on then incr(i);
  $camp\text{-}size \gets 0$;
  while TOW [i] $\neq$ end-of-char do ⟨ Process count for best dyn-f value 69⟩;
  $b\_comp\_size \gets comp\_size$; $dyn\text{-}f \gets 0$;
  for $i \gets 1$ to 13 do
    begin $comp\_size \gets camp\text{-}size + deriv[i]$;
    if $comp\_size \le b\_comp\_size$ then
      begin $b\text{-}camp\text{-}size \gets comp\_size$; $dyn\text{-}f \gets i$;
      end:
    end;
  $camp\text{-}size \gets (b\_comp\_size + 1)$ div 2;
  if $(camp\text{-}size > (height * width + 7)$ div $8)$ V $(height * width = 0)$ then
    begin $comp\_size \gets (height * width + 7)$ div $8$; $dyn\text{-}f \gets 14$;
    end;
  d_print_ln( ´Best␣packing␣is␣dyn_f␣of␣´, dyn-f : 1, ´␣with␣length␣´, camp-size : 1);
  ⟨Write character preamble 71⟩;
  if $dyn\text{-}f \neq 14$ then ⟨ Send compressed format 75⟩
  else if $height > 0$ then ⟨ Send bit map 76 ⟩
```

This code is used in section 62.

69. When we enter this module, we have a count at *row* [i]. First, we add to the *camp-size* the number of nybbles that this count would require, assuming *dyn-f* to be zero. When *dyn-f* is zero, there are no one nybble counts, so we simply choose between two-nybble and extensible counts and add the appropriate value.

Next, we take the count value and determine the value of *dyn-f* (if any) that would cause this count to take either more or less nybbles. If a valid value for *dyn-f* exists in this range, we accumulate this change in the *deriv* array.

One special case handled here is a repeat count of one. A repeat count of one will never change the length of the raster representation, no matter what *dyn-f* is, because it is always represented by the nybble value 15.

```
⟨ Process count for best dyn-f value 69 ⟩ ≡
  begin $j \leftarrow row[i]$;
  if $j = -1$ then incr(comp-size)
  else begin if $j < 0$ then
        begin incr(comp_size); $j \leftarrow -j$;
        end;
    if $j < 209$ then $comp\_size \leftarrow comp\_size + 2$
    else begin $k \leftarrow j - 193$;
      while $k \geq 16$ do
        begin $k \leftarrow k$ div 16; $camp\text{-}size \leftarrow camp\text{-}size + 2$;
        end;
      $camp\text{-}size \leftarrow camp\text{-}size + 1$;
      end;
    if $j < 14$ then decr(deriv[j])
    else if $j < 209$ then incr(deriv[(223 - j) div 15])
      else begin $k \leftarrow 16$;
        while $(k * 16 < j + 3)$ do $k \leftarrow k * 16$;
        if $j - k \leq 192$ then $deriv[(207 - j + k) \text{ div } 15] \leftarrow deriv[(207 - j + k) \text{ div } 15] + 2$;
        end;
    end;
  incr(i);
  end
```

This code is used in section 68.

70. We need a handful of locals:

```
⟨ Locals to pack-and-send-character 65 ⟩ +≡
dyn-f: integer;  { packing value }
height, width: integer;  { height and width of character }
z-offset, y_offset: integer;  { offsets }
deriv: array [1 . . 13] of integer;  { derivative }
b_comp_size: integer;  { best size }
first-on: boolean;  { indicates that the first bit is on }
Jag-byte: integer;  { flag byte for character }
state: boolean;  { state variable }
on : boolean;  { white or black? }
```

71. Now we write the character preamble information. First we need to determine which of the three formats we should use.

```
⟨ Write character preamble 71⟩ ≡
  flag-byte ← dyn_f * 16;
  if first-on then flag_byte ← flag_byte + 8;
  if (gf-ch > 255) ∨ (tfm_width[gf_ch_mod_256] > 16777215) ∨ (tfm-width [gf_ch_mod_256] <
        0) ∨ (dy[gf_ch_mod_256] ≠ 0) ∨ (dx [gf_ch_mod_256] < 0) ∨ (dx[gf_ch_mod_256] mod 65536 ≠
        0) ∨ (camp-size > 196579) ∨ (width > 65535) ∨ (height > 65535) ∨ (x_offset > 32767) ∨ (y_offset >
        32767) ∨ (x_offset < -32768) ∨ (y_offset < -32768) then ⟨Write long character preamble 72⟩
  else if (dx[gf_ch] > 16777215) ∨ (width > 255) ∨ (height > 255) ∨ (x_offset > 127) ∨ ( y_offset >
          127) ∨ (x_offset < -128) ∨ (y_offset < -128) ∨ (camp-size > 1016) then
      ⟨ Write two-byte short character preamble 74⟩
    else ⟨ Write one-byte short character preamble 73⟩
```

This code is used in section 68.

72. If we must write a long character preamble, we adjust a few parameters, then write the data.

```
⟨ Write long character preamble 72⟩ ≡
  begin flag-byte ← flag_byte + 7; pk_byte (flag-byte); comp_size ← comp_size + 28; pk-word (comp_size);
  pk-word (gf-ch); pred_pk_loc ← pk_loc + comp_size; pk-word (tfm-width [gf_ch_mod_256]);
  pk-word (dx [gf_ch_mod_256]); pk-word ( dy [gf_ch_mod_256]); pk-word (width); pk-word (height);
  pk_word(x_offset); pk_word( y-offset);
  end
```

This code is used in section 71.

73. Here we write a short short character preamble, with one-byte size parameters.

```
⟨ Write one-byte short character preamble 73⟩ ≡
  begin comp_size ← comp_size + 8; flag_byte ← flag-byte + comp_size div 256; pk_byte(flag_byte);
  pk_byte( comp_size mod 256); pk_byte(gf_ch); pred_pk_loc ← pk_loc + camp-size;
  pk_three_bytes( tfm_width[gf_ch_mod_256]); pk_byte( dx[gf_ch_mod_256] div 65535); pk_byte( width);
  pk_byte(height); pk_byte(x_offset); pk_byte( y-offset);
  end
```

This code is used in section 71.

74. Here we write an extended short character preamble, with two-byte size parameters.

```
⟨ Write two-byte short character preamble 74⟩ ≡
  begin comp_size ← comp_size + 13; flag-byte ← flag-byte + comp_size div 65536 + 4; pk_byte(flag_byte);
  pk_halfword (comp_size mod 65536): pk_byte(gf_ch); pred_pk_loc ← pk_loc + comp_size;
  pk-three-bytes( tfm-width [gf-ch-mod-2561); pk_halfword (dx [gf_ch_mod_256] div 65536):
  pk-halfword (width); pk-halfword (height ); pk-halfword (x-offset ); pk-halfword (y_offset );
  end
```

This code is used in section 71.

75. At this point, we have decided that the run-encoded format is smaller. (This is almost always the case.) We send out the data, a nybble at a time.

```
⟨ Send compressed format 75 ⟩ ≡
  begin bit-weight ← 16; max-2 ← 208 − 15 ∗ dyn-f; i ← 0;
  if row[i] = 0 then incr(i);
  while row[i] ≠ end-of-char do
    begin j - row[i];
    if j = -1 then pk_nyb(15)
    else begin if j < 0 then
         begin pk_nyb( 14); j ← −j;
         end;
      if j ≤ dyn-f then pk_nyb(j)
      else if j ≤ max-2 then
           begin j ← j − dyn-f − 1; pk_nyb(j div 16 + dyn-f + 1); pk-nyb(j mod 16):
           end
         else begin j ← j − max-2 + 15; k ← 16;
           while k ≤ j do
             begin k - k ∗ 16; pk-nyb(0);
             end;
           while k > 1 do
             begin k ← k div 16; pk_nyb(j div k); j - j mod k;
             end;
           end;
      end;
    incr(i);
    end;
  if bit-weight ≠ 16 then pk-byte(output-byte);
  end
```

This code is used in section 68.

76. This macro is for the case where we have decided to send the character raster packed by bits. It uses the bit counts as well, sending eight at a time. Here we have a miniature packed format interpreter, as we must repeat any rows that are repeated. The algorithm to do this was a lot of fun to generate. Can you figure out how it works?

```
⟨ Send bit map 76⟩ ≡
  begin buff ← 0; p-bit ← 8; i ← 1; h-bit ← width: on ← false; state ← false; count ← row[0];
  repeat-jlag ← 0;
  while (row[i] ≠ end-of-char) ∨ state ∨ (count > 0) do
    begin if state then
      begin count ← r-count; i ← r-i: on ← r-on; decr(repeat_flag);
      end
    else begin r-count ← count; r-i ← i; r-on ← on:
      end;
    ⟨ Send one row by bits 80 ⟩;
    if state ∧ (repeat-flag = 0) then
      begin count ← s-count; i ← s-i; on ← s-on; state ← false;
      end
    else if ¬state ∧ (repeat_flag > 0) then
        begin s-count ← count; s-i ← i; s-on ← on: state ← true;
        end;
    end:
  if p-bit ≠ 8 then pk_byte(buff);
  end
```

This code is used in section 68.

77. All of the remaining locals:

```
⟨ Locals to pack-and-send-character 65⟩ +≡
comp_size: integer;  { length of the packed representation in bytes }
count: integer;  { number of bits in current state to send }
p-bit: integer;  { what bit are we about to send out? }
r-on, s-on: boolean;  { state saving variables }
r-count, s-count: integer;  { ditto}
r-i, s-i: integer;  {and again. }
max_2: integer;  { the highest count that fits in two bytes }
```

78. We make the *power* array global.

```
⟨Globals in the outer block 11⟩ +≡
power: array [0 . . 8] of integer;  { easy powers of two }
```

79. We initialize the power array.

```
⟨ Set initial values 12⟩ +≡
  power[0] ← 1;
  for i ← 1 to 8 do power[i] ← power[i − 1] + power[i − 1];
```

80. Here we are at the beginning of a row and simply output the next *width* bits. We break the possibilities up into three cases: we finish a byte but not the row, we finish a row. and we finish neither a row nor a byte. But, first, we insure that *we* have a *count* value.

```
⟨ Send one row by bits 80⟩ ≡
  repeat if count = 0 then
        begin if row[i] < 0 then
          begin if ¬state then repeat_flag ← -row[i];
          incr(i);
          end;
        count ← row[i]; incr(i); on ← ¬on;
        end;
     if (count ≥ p_bit) ∧ (p_bit < h_bit) then
        begin { we end a byte, we don't end the row }
        if on then buff ← buff + power[p_bit] − 1;
        pk_byte( buff ); buff ← 0; h_bit ← h_bit − p_bit; count ← count − p_bit; p_bit ← 8;
        end
     else if (count < p_bit) ∧ (count < h_bit) then
          begin { we end neither the row nor the byte }
          if on then buff ← buff + power[p_bit] − power[p_bit − count];
          p_bit ← p_bit − count; h_bit ← h_bit − count; count ← 0;
          end
        else begin { we end a row and maybe a byte }
          if on then buff ← buff + power[p_bit] − power[p_bit − h_bit];
          count ← count − h_bit; p_bit ← p_bit − h_bit; h_bit ← width;
          if p_bit = 0 then
            begin pk_byte(buff); buff ← 0; p_bit ← 8;
            end;
          end;
  until h_bit = width
```

This code is used in section 76.

81. Now we are ready for the routine that writes the preamble of the packed file.

```
  define preamble_comment ≡ ´GFtoPK␣2.0␣output´
  define comm_length = 17
⟨ Write preamble 81⟩ ≡
  pk_byte(pk_pre); pk_byte(pk_id); pk_byte( comm_length);
  for i ← 1 to comm_length do pk_byte(xord[comment [i]]);
  pk_word (design_size); pk_word (check_sum); pk_word (hppp); pk_word (vppp)
```

This code is used in section 51.

82. Of course, we need an array to hold the comment.

```
⟨ Globals in the outer block 11⟩ +≡
comment: packed array [1 . . comm_length] of char;
```

83. ⟨ Set initial values 12⟩ +≡

```
  comment ← preamble_comment ;
```

84. Writing the postamble is even easier.

```
⟨ Write postamble 84⟩ ≡
  pk-byte(pk-post);
  while (pk_loc mod 4 ≠ 0) do pk_byte(pk_no_op)
```

This code is used in section 51.

85. Once we are finished with the GF file, we check the status of each character to insure that each character that had a locater also had raster information.

```
⟨ Check for un-rasterized locaters 85⟩ ≡
  for i ← 0 to 255 do
    if status[i] = located then print-ln( ´Character␣´, i : 1, ´␣missing␣raster␣information!´)
```

This code is used in section 86.

86. Finally, the main program.

```
  begin initialize; convert_gf_file; ⟨ Check for un-rasterized locaters 85 ⟩;
  print-ln (gf-Zen : 1, ´␣bytes␣packed␣to␣´, pk_loc : 1, ´␣bytes.´);
final-end: end.
```

87. A few more globals.

```
⟨ Globals in the outer block 11⟩ +≡
check-sum: integer;   { the checksum of the file }
dir-ptr: integer;   { where does the directory information start? }
design-size: integer;   { the design size of the font }
h-mug: integer;   { the pixel magnification in pixels per inch }
i: integer;
```

88. System-dependent changes. This section should be replaced, if necessary, by changes to the program that are necessary to make GFtoPK 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.

89. Index. Pointers to error messages appear here together with the section numbers where each identifier is used.

The GFtoDVI processor

(Version 2.0, April 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␣2.0'   { 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 = 1000;  { 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 }
```

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 donel = 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-lot − 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 136.

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. They have been considerably simplified, since GFtoDVI need not deal with the controversial ASCII codes less than '40.

⟨ Types in the outer block 9 ⟩ +≡
ASCII_code = "␣" . . "~"; { 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 lower case letters. Nowadays, of course, we need to deal with both upper and lower case alphabets 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 "~").

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* = 127 { 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** [0 . . 255] **of** *text_char*; { specifies conversion of output characters }

See also sections 15, 17, 18, 37, 46, 48, 49, 53, 71, 76, 80, 86, 87, 93, 96, 102, 105, 117, 124, 131, 137, 146, 152, 155, 157, 163, 165, 171, 179, 180, 204, 208, 209, and 217.

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 anv svstem-dependent changes.

```
⟨ Set initial values 13⟩ ≡
  for i ← 0 to '37 do xchr[i] ← '?';
  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] ← '0';
  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] t '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] ← '~';
  for i ← '177 to 255 do xchr[i] ← '?';
```

See also sections 14, 54, 97, 103, 106, 123, and 139.

This code is used in section 3.

14. 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 lust-text-char do xord[chr(i)] ← "␣";
  for i ← "!" to "~" do xord[xchr[i]] ← i;
```

15. 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 }
```

16. 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) A ¬ eoln ( term-in) do
    begin buffer [line-length] ← xord [ term-in ↑]; incr (line-length); get (term-in);
    end;
  end;
```

17. ⟨ Globals in the outer block 12 ⟩ +≡
line-length: 0 . . terminal-line-length; { end of line read by *input_ln* }

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 }

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-l 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-O,* except that character number c is typeset. TEX82 uses this command for characters in the range $128 \leq c < 256$.

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

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 $a \leq 0$ or $b \leq 0$, 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 \ countO ... `\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.

rop 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, $-128 \leq b < 128$; 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 $-32768 \leq b < 32768$.

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

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. (Some day we will set $i = 3$, when DVI format makes another incompatible change-perhaps in 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 *mug* parameter is what TEX82 calls **\mag**, i.e., 1000 times the desired magnification. The actual fraction by which dimensions are multiplied is therefore $mn/1000d$. 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 mug in the preamble and postamble. (Fancy DVI-reading programs allow users to override the *mug* 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 $0 \leq k < 256$.

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

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

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 it is given in DVI units that have not been corrected for the magnification *mug* found in the preamble. Thus, font k is to be used at *mug* . $s/1000d$ 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 and xxx 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

$$\begin{array}{l} \textit{post}\ p[4]\ num[4]\ \textit{den}\ [4]\ mag[4]\ l[4]\ u[4]\ s[2]\ t[2] \\ \langle\text{ font definitions }\rangle \\ \textit{post-post}\ q[4]\ i[1]\ 223\text{'s}[\geq 4] \end{array}$$

Here p is a pointer to the final *bop* in the file. The next three parameters, num, *den.* and *mug,* 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 discovers 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 METRFONT; 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 METRFONT.

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 METRFONT? 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 METRFONT 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 METRFONT 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 *puint* command with d = 0: it does nothing but change the *paint-switch* from *black* to *white* or vice versa.

paint-l 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.

puintl 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 *paintl* , but d can be as high as 65535.

paint3 66 $d[3]$. Same as *paintl* , 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] *mux-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 $\le m \le$ mux-m and min-n $\le$ n $\le$ mux-n. (The values of *mux-m* and min-n need not be the tightest bounds possible.) When a GF-reading program *sees* a *boc*, it can *use min-m. mux-m, min-n,* and *mux-n* to initialize the bounds of an array. Then it sets $m \leftarrow$ *min-m,* $n \leftarrow$ *max_n*, and *paint-switch* $\leftarrow$ *white.*

bocl 68 c[1] *del_m*[1] mux-m[1] *del_n*[1] *max_n*[1]. Same as *boc*, but p is assumed to be -1: also *del_m* = *mux-m* − *min-m* and *del_n* = *mux-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 + *min-m, paint-switch* $\leftarrow$ *white.* (This finishes one row and begins anot her, ready to whiten the left most pixel in the new row.

skipl 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* ***skipl*** , but d can be as large as 65535.

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

new-row-O 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-l through *new-row-164* (opcodes 75 to 238). Same as *new-row-O,* but with $m \leftarrow$ *min-rn + 1* through *min-m* + 164, respectively.

xxx1 2 3 9 $k[1]$ $x[k]$. This command is undefined in general: it functions as a *(k* + 2)-byte *no-op* unless special GF-reading programs are being used. METRFONT 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 anywhere 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-Zoc 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 ≤ *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 *puin t-0* = 0 { beginning of the *paint* commands }
define *paint 1* = 64 { move right a given number of columns, then black ↔ white }
define *pain t2* = 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*$[4]$ *mux-m* $[4]$ *min_n*$[4]$ *mux-n* $[4]$
> ⟨ character locators ⟩
> *post-post* $q[4]$ $i[1]$ 223's$[\geq 4]$

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 *uppp* 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, mux-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; *mux-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 $\leq$ -50. If there are no characters in the file, it's possible to have *min-m* > *mux-m* and/ or *min-n* > *max_n*.)

32. Character locators are introduced by *char-Zoc* 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^{24} 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 postarnble, 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). METRFONT 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. METRFONT 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 METRFONT 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. **GFt oDVI** has default fonts that will be used if none other are specified; the "current" title font is initially the default title font.

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

t itlef **ontat** 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.

labelf ontat *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 $bc - 1 \le ec \le 255$, $ne \le 256$, and

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

Note that a font may contain as many as 256 characters (if $bc = 0$ and $ec = 255$), and as few as 0 characters (if $bc = ec + 1$). 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

$$\begin{aligned}
\textit{header}&: \text{array } [0\,..\,lh-1] \text{ of } \textit{stuff}\\
\textit{char-info}&: \text{array } [bc\,..\,ec] \text{ of } \textit{char-info-word}\\
\textit{width}&: \text{array } [0\,..\,nw-1] \text{ of } \textit{fix-word}\\
\textit{height}&: \text{array } [0\,..\,nh-1] \text{ of } \textit{fix-word}\\
\textit{depth}&: \text{array } [0\,..\,nd-1] \text{ of } \textit{fix-word}\\
\textit{italic}&: \text{array } [0\,..\,ni-1] \text{ of } \textit{fix-word}\\
\textit{lig\_kern}&: \text{array } [0\,..\,nl-1] \text{ of } \textit{lig-kern-command}\\
\textit{kern}&: \text{array } [0\,..\,nk-1] \text{ of } \textit{fix-word}\\
\textit{exten}&: \text{array } [0\,..\,ne-1] \text{ of } \textit{extensible-recipe}\\
\textit{param}&: \text{array } [1\,..\,np] \text{ of } \textit{fix-word}
\end{aligned}$$

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 one of the *fix-word* values will 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 lo-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 *tug* (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 correct ions.

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 *tug* field in a *char-info-word* has four values that explain how to interpret the *remainder* field.

tug = 0 (no-tug) means that *remainder* is unused.
tug = 1 *(lig-tug)* means that this character has a ligature/ kerning program starting at *Zig-kern [remainder].*
tug = 2 (list-tug) 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.
tug = 3 (ext-tug) 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-tug = 0 { vanilla character }
define *lig-tug* = 1 { character has a ligature/ kerning program }
define *list-tug* = 2 { character has a successor in a charlist }
define ext- *tug* = 3 { character is extensible }

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

first byte: *stop-bit,* indicates that this is the final program step if the byte is 128 or more.
second byte: *next-char,* "if *next-char* follows the current character, then perform the operation and stop, otherwise continue."
third byte: *op_bit*, indicates a ligature step if less than 128, a kern step otherwise.
fourth byte: *remainder.*

In a ligature step the current character and *next-char* are replaced by the single character whose code is *remainder.* In a kern step, an additional space equal to *kern*[*remainder*] *is* inserted between the current character and *next-char.* (The value of *kern*[*remainder*] is often negative, so that the characters are brought closer together by kerning; but it might be positive.)

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-lot* 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); reud(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-Zoc): 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-Zoc + 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-quud ← (((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-quurterword* = -128 and *mux-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 mux-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
```

⟨Types in the outer block 9⟩ +≡

```
  quarterword = min-quarterword . . mux-quarterword;  {1/4 of a word}
  four-quarters = packed record b0: quarterword;
    bl : 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 [0 . . *font-mem-size]* of *memory-word;* { the font metric data}
fmem-ptr : *0* . . *font-mem-size;* { 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 }

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

$$b = height\text{-}index \times 16 + depth\text{-}index,$$

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-tug(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 (t. 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 *stop-bit* (#) ≡ #. *b0*
define *next-char* (#) ≡ #. *b1*
define *op-bit(#)* ≡ #.*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 *Zig-kern-start(#)* ≡ ***lig_kern_base***[#] + *rem-byte* { beginning of lig/kern program }
define *char-kern-end(#)* ≡ *rem-byte(#)*] *.sc*
define *char-kern(#)* ≡ *font-info [* ***kern_base***[#] + *char-kern-end*

5 7 . 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 bud-tfm = 11 { label for read-font-info }
define ubend ≡ goto bud-tfm { do this when the TFM data is wrong}
procedure read-font-info(f : integer; s : scaled); {input a TFM file}
  label done, bud- tfm ;
  var k: 0 . . font-mem-size; { index into font-info }
    lf, Zh, bc, ec, nw, nh, nd, ni, nl, nk, ne, np: 0 . . 65535; {sizes of subfiles}
    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; ubend if the TFM file is malformed; otherwise goto done 59 ⟩;
bud-tfm: print-nZ( ´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: *ubend* 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* + *bl* ; *rend-two-halves-end*

⟨ Read the TFM size fields 60 ⟩ ≡

```
  begin read_two_halves(lf)(lh); read-two-huZves( bc)( ec);
  if (bc > ec + 1) ∨ (ec > 255) then ubend;
  read_two_halves(nw)(nh); read_two_halves(nd)(ni); read-two-huZves(nZ)(nk); read_two_halves(ne)(np);
  if lf ≠ 6 + lh + (ec − bc + 1) + nw + nh + nd + ni + nl + nk + ne + np then ubend:
  end
```

This code is used in section 59

61. The preliminary settings of the index variables *width-base, Zig-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 − Zh;  { 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␣f or␣TFM␣f ile ! ');
  char-buse [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; Zig-kern-buse[f] ← italic_base[f] + ni;
  kern-base [f] ← Zig-kern-buse[f] + nl; exten_base[f] ← kern_base[f] + nk;
  purum-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 GFt oDVI.

```
  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 ubend;
  store-four-quurters(font-check[f]);    read-tfm-word;
  if b0 > 127 then ubend;  { design size must be positive }
  z ← ((b0 * 256 + b1) * 256 + b2) * 16 + (b3 div 16);
  if z < unity then ubend:
  while lh > 2 do
    begin read-tfm-word; decr (Zh);  { 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
    Zig-tug: if b3 ≥ nl then ubend;
    ext-tug: if b3 ≥ ne then ubend;
    no-tug, list-tug: 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*, bl, *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 ubend;   { 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 ubend;   { depth[0] must be zero }
  if font_info[italic_base[f]].sc ≠ 0 then ubend;   { 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) V (# > ec) then ubend
        end
```

```
⟨ Read ligature/kern program 66 ⟩ ≡
  begin for k ← lig_kern_base[f] to kern_base[f] − 1 do
    begin store-four-qwwters(font-info[k].qqqq); check-byte-runge( b1 );
    if b2 < 128 then check-byte-runge( b3)   { check ligature }
    else if b3 ≥ nk then ubend;   { check kern }
    end;
  if (nl > 0) A (b0 < 128) then ubend;   { check for stop bit on last command }
  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 $k \leftarrow exten\_base[f]$ to $param\_base[f] - 1$ do
  begin $store\_four\_quarters(font\_info[k].qqqq)$;
  if $b0 \neq 0$ then $check\_byte\_range(b0)$;
  if $b1 \neq 0$ then $check\_byte\_range(b1)$;
  if $b2 \neq 0$ then $check\_byte\_range(b2)$;
  $check\_byte\_range(b3)$;
  end
```

This code is used in section 59.

68. ⟨ Read font parameters 68⟩ ≡

```
begin for $k \leftarrow 1$ to $np$ do
  if $k = 1$ then { the $slant$ parameter is a pure number }
    begin $read\_tfm\_word$;
    if $b0 > 127$ then $sw \leftarrow b0 - 256$ else $sw \leftarrow b0$;
    $sw \leftarrow sw * '400 + b1$; $sw \leftarrow sw * '400 + b2$; $font\_info[param\_base[f]].sc \leftarrow (sw * '20) + (b3$ div $'20)$;
    end
  else $store\_scaled(font\_info[param\_base[f] + k - 1].sc)$;
for $k \leftarrow np + 1$ to 8 do $font\_info[param\_base[f] + k - 1].sc \leftarrow 0$;
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 $adjust(\#) \equiv \#[f] \leftarrow qo(\#[f])$ { correct for the excess $min\_quarterword$ that was added }

⟨ Make final adjustments and **goto** *done* 69⟩ ≡
$font\_bc[f] \leftarrow bc$; $font\_ec[f] \leftarrow ec$; $adjust(width\_base)$; $adjust(lig\_kern\_base)$; $adjust(kern\_base)$; $adjust(exten\_base)$; $decr(param\_base[f])$; $fmem\_ptr \leftarrow fmem\_ptr + lf$; **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 . . mux-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 valus 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-p tr : 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 1 more characters to *str-pool,* we shall write *str-room(Z),* 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 = mux-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-strl (#) ≡ buffer[1] ← #; init_str0
  define init_str2 (#) ≡ buffer[2] ← #; init-strl
  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 'off set' }
define x_offset_code = 17   { code for the keyword 'xoff set' }
define y-onset-code = 18   { code for the keyword 'yoff set' }
define max.-keyword = 18   { largest keyword code number }
```

```
⟨ Initialize the strings 77 ⟩ ≡
  str_ptr ← 0; poobptr ← 0; str-start [0] + 0;
  1 ← 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);
  1 ← 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);
  1 ← 11; init-strll ("t")("i")("t")("l")("e")("f")("o")("n")("t")("a")("t")(title_font + ut-code);
  l ← 11; init-strll ("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-strll ("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);
  1 ← 13;
  init_str13("r")("u")("l")("e")("t")("h")("i")("c")("k")("n")("e")("s")("s")(rule_thickness_code);
  1 ← 6; init_str6("o")("f")("f")("s")("e")("t")(offset_code);
  1 ← 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-onset-code);
```

See also sections 78 and 88.

This code is used in section 216.

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 METRFONT 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 ← 3; *init_str3*(".")("g")("f")(*gf_ext*);
l ← 4; *init_str4*(".")("d")("v")("i")(*dvi_ext*);
l ← 4; *init_str4*(".")("t")("f")("m")(*tfm-ext*);
l ← 7: *init_str7*("␣")("␣")("P")("a")("g")("e")("␣")(*page_header*);
l←12; *init_str12*("␣")("␣")("C")("h")("a")("r")("a")("c")("t")("e")("r")("␣")(*char_header*);
l ← 6; *init_str6*("␣")("␣")("E")("x")("t")("␣")(*ext_header*);
l ← 4; *init_str4*("␣")("␣")("`")("`")(*left_quotes*);
l ← 2; *init_str2*("´")("´")(*right_quotes*);
l f- 3; *init_str3*("␣")("=")("␣")(*equals-sign*);
1 ← 4; *init_str4*("␣")("+")("␣")("(")(*plus_sign*);
1 ←4; *init_str4*("c")("m")("r")("8")(*default-title-font*);
1 ← 6; *init_str6*("c")("m")("t")("t")("1")("0")(*default_label_font*);
l ← 4; *init_str4*("g")("r")("a")("y")(*default-gray-font*);
l ← 5; *init_str5*("l")("o")("g")("o")("8")(*logo-font-name*);
l ← 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 $xxx1 \le cur\text{-}gf \le no\text{-}op$. 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 donel;
  if j = longest-keyword then goto not-found:
  incr(j); buffer[j] ← get-byte;
  if buffer[j] = "␣" then goto donel;
  end;
donel : ⟨ 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) = 1 then
    begin nl ← 0; n2 ← str_start[m];
    while (nl < l) ∧ (buffer [nl + 1] = str_pool[n2]) do
      begin incr (nl ); incr (n2);
      end;
    if nl = 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 'TeXf onts : '.

```
⟨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-sturt [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-pooZ[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 rnodified 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, nullstring, 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 user 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-nume;
  for k ← title-font to logo-font do
    begin font-urea [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 134⟩;
  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 rnultiples 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(O, 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 [fontarea[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 t 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 216.

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.*

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.

```
procedure hbox(s : str_number; f : internal_font_number; send-it : boolean):
  label continue, done;
  var k, max-k: pool-pointer; { indices into str-pool }
    i, j: four-quarters; { font information words }
    c: eight-bits; { a character code }
    r: quarterword; { ligature or kern must match this }
    l: 0 . . font-mem-size; {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 }
  begin box-width ← 0; box-height ← 0; box-depth ← 0;
  k ← str_start[s]; max_k ← str_start [s + 1];
  while k < max_k do ⟨ Typeset character str_pool[k], possibly making a ligature with the following
        character or characters, and advance k 118⟩;
  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 }
```

118. ⟨Typeset character *str_pool*[*k*], possibly making a ligature with the following character or characters, and advance *k* 118 ⟩ ≡

```
  begin c ← str_pool[k]; incr(k); kern-amount ← 0;
  if c = "␣" then kern-amount ← space(f)
  else if c ≥ font_bc[f] then
      if c ≤ font_ec[f] then
        begin continue: i ← char_info(f)(c);
        if char-exists(i) then
          begin if char-tag(i) = lig-tag then
            if k < max-k then
              ⟨ Look for possible ligature or kern: goto continue if c has been replaced by a ligature 119⟩;
          ⟨ Typeset character c 120⟩;
          end;
        end;
  if kern-amount ≠ 0 then
    begin box-width ← box-width + kern-amount;
    if send-it then
      begin dvi_out (right4 ); dvi_four(kern_amount);
      end:
    end:
  end
```

This code is used in section 116.

119. ⟨Look for possible ligature or kern: **goto** *continue* if c has been replaced by a ligature 119⟩ ≡

```
  begin $r \gets qi(str\_pool[k]);\ l \gets lig\_kern\_start(f)(i)$;
  repeat $j \gets font\_info[l].qqqq$;
    if $next\_char(j) = r$ then
      if $op\_bit(j) < qi(kern\_flag)$ then
        begin $c \gets qo(rem\_byte(j));\ incr(k)$; goto continue;
        end
      else begin $kern\_amount \gets char\_kern(f)(j)$; goto done;
        end;
    $incr(l)$;
  until $stop\_bit(j) \ge qi(stop\_flag)$;
done: end
```

This code is used in section 118.

120. ⟨Typeset character c 120⟩ ≡

```
  $box\_width \gets box\_width + char\_width(f)(i);\ hd \gets height\_depth(i);\ x \gets char\_height(f)(hd)$;
  if $x > box\_height$ then $box\_height \gets x$;
  $x \gets char\_depth(f)(hd)$;
  if $x > box\_depth$ then $box\_depth \gets x$;
  if $send\_it$ then $typeset(c)$;
```

This code is used in section 118.

121. 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.

122. 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.

123. 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:

```
blank  ←— baseline
  x
  x
blank
  x
```

(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 X000000. xxooooo, . . . , XXXXXXO, XXXXXXX of length seven; positions 71-78 are, similarly, reserved for the length-eight patterns X0000000 through XXXXXXXX. The length-nine patterns X00000000 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 125⟩;
  for k ← 7 to 12 do ⟨ Add special k-bit characters of the form X..XO..O 126⟩;
```

```
124. ⟨ 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 }
```

```
125. ⟨Add a full set of k-bit characters 125⟩ ≡
  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 123.

126. ⟨ Add special k-bit characters of the form X. .XO. .O 126⟩ ≡

```
begin $n \leftarrow two\_to\_the[k - 1]$;
for $j \leftarrow k$ **downto** 1 do
  begin $incr(m)$; $d[m] \leftarrow n + n$:
  if $j = k$ then $c[m] \leftarrow n$
  else $c[m] \leftarrow c[m - 1] + two\_to\_the[j - 1]$;
  end:
end
```

This code is used in section 123.

127. Position 0 of a gray font is reserved for the "dot" character, which should have positive height IL' 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 + δ, $y + \epsilon$) for $-w' \leq \delta \leq$ w' and $-h' \leq \epsilon \leq h'$; the rectangular box for a label will butt up against the rectangle enclosing the dot.

128. 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 l-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 $1 \leq k \leq 120$ 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 l-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.

129. 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. rat her 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.

130. 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.

131. 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 $[m/q]$ 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 loo-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 }
```

132. 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.

133. 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.

```
134. ⟨ Initialize global variables that depend on the font data 134 ⟩ ≡
  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 166, 172, 181, 202, and 203.

This code is used in section 98.

135. 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:
```

136. 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;
```

137.
```
⟨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 }
```

138. 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;
```

139. 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;
```

140. 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;
```

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

```
⟨ Initialize variables for the next character 141⟩ ≡
  max-node + 0; next [0] ← end-of-list; prev[end_of_list] ← 0; max-height ← 0; max-depth + 0;
```

See also sections 153 and 158.

This code is used in section 216.

142. 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 143 ⟩;
  ⟨ Look for overlaps in node q and its predecessors 144 ⟩;
  overlap + false;
exit: end;
```

143.

```
⟨ Look for overlaps in the successors of node q 143⟩ ≡
  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 142.

144. ⟨ Look for overlaps in node q and its predecessors 144⟩ ≡

```
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 142.

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

$$\begin{array}{ll} info[p] < 0; & p \ge first\text{-}dot; \\ xl[p] = xx[p] - dot\text{-}width, & xr[p] = xx[p] + dot\text{-}width; \\ yt[p] = yy[p] - dot\text{-}height, & yb[p] = yy[p] + dot\text{-}height. \end{array}$$

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

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

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 $d(q,p) < d0$.

146. ⟨ 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? }
```

147. 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 148⟩;
  ⟨ Search for the nearest dot in nodes preceding p 149⟩;
  nearest-dot ← best-q;
  end;
```

148. ⟨ Search for the nearest dot in nodes following p 148 ⟩ ≡

```
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:
   4 ← next [q];
   end
```

This code is used in section 147.

149. ⟨ Search for the nearest dot in nodes preceding p 149 ⟩ ≡

```
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 147.

150. 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.

151. 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 151 ⟩ ≡
  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 156⟩;
  offset-code: ⟨Override the offsets 154⟩;
  x_offset_code: x-offset ← get-yyy;
  y_offset_code: y_offset ← get-yyy;
  title-code: ⟨Store a title 159⟩;
  null-string: ⟨Store a label 160⟩;
  end; { there are no other cases }
  end
```

This code is used in section 216.

152. 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 }
```

153.

```
⟨Initialize variables for the next character 141⟩ +≡
  rule-thickness ← 0; offset_x ← 0; offset_y ← 0; x_offset ← 0; y-offset ← 0; pre-minx ← '2000000000 ;
  pre_max_x ← − '2000000000 ; pre_min_ y ← '2000000000 ; pre_max_ y ← − '2000000000 ;
```

154.

```
⟨ Override the offsets 154⟩ ≡
  begin offset_x ← get-yyy; offset_y ← get-yyy;
  end
```

This code is used in section 151.

155. 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 METRFONT pixels) }
define xl ≡ 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 }

156. ⟨Store a rule 156⟩ ≡

```
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 xl [p] < pre-min-x then pre-min-x + xl [p];
if xl [p] > pre-max-x then pre-max-x ← xl [p];
if yl [p] < pre-min-y then pre-min-y ← yl [p];
if yl [p] > pre_max_ y then pre-max-y ← yl [p];
rule-size [p] ← rule- thickness ;
end
```

This code is used in section 151.

157. 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 }

158. We must start the lists out empty.

⟨ Initialize variables for the next character 141⟩ +≡

```
rule-ptr ← null; title-head ← null; title-tail + null; label-head ← null; label-tail + end-of-list:
first-dot + max_labels;
```

159. ⟨Store a title 159⟩ ≡

```
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 151.

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

```
⟨Store a label 160⟩ ≡
  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 151.

161. 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 161⟩ ≡
  begin check-fonts; ⟨Finish reading the parameters of the boc 162⟩;
  ⟨Get ready to convert METAFONT coordinates to DVI coordinates 167⟩;
  ⟨Output the bop and the title line 169⟩;
  print(' [', total-pages : 1); update-terminal; { print a progress report }
  ⟨Output all rules for the current character 170⟩;
  ⟨Output all labels for the current character 178⟩;
  do-pixels; dvi_out (eop); { finish the page }
  ⟨Adjust the maximum page width 200⟩;
  print (']'); update-terminal;
  end
```

This code is used in section 216.

```
162. ⟨Finish reading the parameters of the boc 162⟩ ≡
  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 161.

163. ⟨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* }

164. 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 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;
```

165. ⟨ 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 }

166. ⟨Initialize global variables that depend on the font data 134⟩ +≡

```
  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;
```

167. ⟨ Get ready to convert METAFONT coordinates to DVI coordinates 167⟩ ≡

```
  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 161.

168. 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;
```

169. ⟨Output the *bop* and the title line 169⟩ ≡

```
  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 (lust-bop); lust-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-stump, 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-heud ≠ 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-heud ← next [title-head];
    until title-head = null;
    end:
  dvi-out (pop)
```

This code is used in section 161.

```
170.  define tol ≡ 6554  { one tenth of a point. in DVI coordinates }
⟨ Output all rules for the current character 170⟩ ≡
  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 173⟩
      else if abs( temp-y − dvi-y) < tol then ⟨ Output a horizontal rule 174⟩
        else ⟨ Try to output a diagonal rule 175⟩;
      end;
    end
```
This code is used in section 161.

```
171.  ⟨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 }
```

```
172.  ⟨Initialize gl ob al variables that depend on the font data 134 ⟩ +≡
  gra y-rule- thickness ← default-rule- thickness (gru y-font );
  if gray-rule-thickness = 0 then gray-rule-thickness + 26214;  { 0.4 pt }
```

```
173.  ⟨ Output a vertical rule 173⟩ 3
  begin if temp- y > dvi-y then
    begin k ← temp-y; temp-y ← dvi-y; dvi-y ← k;
    end;
  dvi_go to( 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 170.

```
174.  ⟨ Output a horizontal rule 174 ⟩ ≡
  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 170.

175. ⟨ Try to output a diagonal rule 175⟩ ≡

```
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 176⟩;
    ⟨ Vertically typeset p copies of character k + 1 177⟩;
    dvi_out(pop);
    end;
  end
```

This code is used in section 170.

176. ⟨Vertically typeset q copies of character k 176⟩ ≡

```
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 175.

177. ⟨ Vertically typeset p copies of character $k + 1$ 177⟩ ≡

```
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 175.

178. 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 178⟩ ≡

```
overflow_line ← 1;
if label_head ≠ null then
  begin next[label_tail] ← null; select_font(gray_font); ⟨ Output all dots 184⟩;
  ⟨ Find nearest dots, to help in label positioning 188⟩;
  select_font(label_font); ⟨ Output all prescribed labels 186⟩;
  ⟨Output all attachable labels 190⟩;
  ⟨ Output all overflow labels 197⟩;
  end
```

This code is used in section 161.

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

180. 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 }
```

181. ⟨Initialize global variables that depend on the font data 134⟩ +≡

```
  i ← 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;
```

182. 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] ← dvi_x − (box_width div 2); xl[p] ← xx[p] − delta; xr[p] ← xx[p] + box_width + delta;
  yb[p] ← dvi_y − dot_height; yy[p] ← yb[p] − box_depth; yt[p] ← yy[p] − box_height − delta;
  end;
```

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

```
procedure bot_coords(p : node_pointer);
  begin $xx[p] \gets dvi\_x - (box\_width$ div $2)$; $xl[p] \gets xx[p] - delta$; $xr[p] \gets xx[p] + box\_width + delta$;
  $yt[p] \gets dvi\_y + dot\_height$; $yy[p] \gets yt[p] + box\_height$; $yb[p] \gets yy[p] + box\_depth + delta$;
  end;
procedure right_coords(p : node_pointer);
  begin $xl[p] \gets dvi\_x + dot\_width$; $xx[p] \gets xl[p]$; $xr[p] \gets xx[p] + box\_width + delta$;
  $yy[p] \gets dvi\_y + half\_x\_height$; $yb[p] \gets yy[p] + box\_depth + delta$; $yt[p] \gets yy[p] - box\_height - delta$;
  end;
procedure left_coords(p : node_pointer);
  begin $xr[p] \gets dvi\_x - dot\_width$; $xx[p] \gets xr[p] - box\_width$; $xl[p] \gets xx[p] - delta$;
  $yy[p] \gets dvi\_y + half\_x\_height$; $yb[p] \gets yy[p] + box\_depth + delta$; $yt[p] \gets yy[p] - box\_height - delta$;
  end;
```

184. ⟨ Output all dots 184 ⟩ ≡

```
  $p \gets label\_head$; $first\_dot \gets max\_node + 1$;
  while $p \neq null$ do
    begin $convert(xx[p], yy[p])$; $xx[p] \gets dvi\_x$; $yy[p] \gets dvi\_y$;
    if $lab\_typ[p] <$ "5" then ⟨Enter a dot for label $p$ in the rectangle list, and typeset the dot 185⟩;
    $p \gets next[p]$;
    end
```

This code is used in section 178.

185. 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 $dot\_for\_label \equiv xl$
  define $label\_for\_dot \equiv info$
```

⟨Enter a dot for label p in the rectangle list, and typeset the dot 185⟩ ≡

```
  begin $q \gets get\_avail$; $dot\_for\_label[p] \gets q$; $label\_for\_dot[q] \gets p$;
  $xx[q] \gets dvi\_x$; $xl[q] \gets dvi\_x - dot\_width$; $xr[q] \gets dvi\_x + dot\_width$;
  $yy[q] \gets dvi\_y$; $yt[q] \gets dvi\_y - dot\_height$; $yb[q] \gets dvi\_y + dot\_height$;
  $node\_ins(q, 0)$;
  $dvi\_goto(xx[q], yy[q])$; $dvi\_out(0)$; $dvi\_out(pop)$;
  end
```

This code is used in section 184.

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

⟨ Output all prescribed labels 186⟩ ≡

```
  $q \gets end\_of\_list$;  { $label\_head = next[q]$ }
  while $next[q] \neq null$ do
    begin $p \gets next[q]$;
    if $lab\_typ[p] >$ "0" then
      begin $next[q] \gets next[p]$;
      ⟨ Enter a prescribed label for node $p$ into the rectangle list, and typeset it 187⟩;
      end
    else $q \gets next[q]$;
    end
```

This code is used in section 178.

187. ⟨ Enter a prescribed label for node p into the rectangle list, and typeset it 187⟩ ≡

```
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 186.

188. 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 188 ⟩ ≡

```
p + label-head;
while p ≠ null do
  begin if lab_typ[p] ≤ "0" then ⟨ Compute the octant code for floating label p 189 ⟩;
  p ← next [p];
  end;
```

This code is used in section 178.

189. 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 189⟩ ≡
  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 188.

190. 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 190⟩ ≡
  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 178.

191. 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 192⟩;
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:
```

192. ⟨ Find non-overlapping coordinates, if possible, and **goto** found; otherwise set *place-label* ← *false* and return 192⟩ ≡

```
dfl ← dot-for-label[p]; oct ← octant[p]; ⟨ Try the first choice for label direction 193⟩;
⟨ Try the second choice for label direction 194⟩;
⟨ Try the third choice for label direction 195⟩;
⟨ Try the fourth choice for label direction 196⟩;
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 191.

193. ⟨ Try the first choice for label direction 193⟩ ≡

```
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 192.

194. ⟨Try the second choice for label direction 194⟩ ≡

```
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 192.

195. ⟨ Try the third choice for label direct ion 195 ⟩ ≡

```
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 192.

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

```
case oct of
fzrst-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 192.

197. ⟨ Output all overflow labels 197⟩ ≡

```
⟨ Remove all rectangles from list, except for dots that have labels 198⟩;
$p \leftarrow label\text{-}head$;
while $p \neq null$ do
  begin ⟨ Typeset an overflow label for $p$ 199 ⟩;
  $p \leftarrow next[p]$;
  end
```

This code is used in section 178.

198. 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 198⟩ ≡
  $p \leftarrow next[0]$;
  while $p \neq end\text{-}of\text{-}list$ do
    begin $q \leftarrow next[p]$;
    if $(p < first\text{-}dot) \lor (label\_for\_dot[p] = null)$ then
      begin $r \leftarrow prev[p]$; $next[r] \leftarrow q$; $prev[q] \leftarrow r$; $next[p] + r$:
      end;
    $P \leftarrow q$;
    end
```

This code is used in section 197.

199. Now we have to insert p into the list temporarily, because of the way *nearest-dot* works.

```
⟨ Typeset an overflow label for $p$ 199⟩ ≡
  begin $r \leftarrow next[dot\_for\_label[p]]$; $s \leftarrow next[r]$; $t \leftarrow next[p]$; $next[p] + s$; $prev[s] \leftarrow p$; $next[r] \leftarrow p$;
  $prev[p] \leftarrow r$;
  $q \leftarrow nearest\text{-}dot(p, 0)$;
  $next[r] \leftarrow s$; $prev[s]$ - $r$; $next[p] \leftarrow t$;    { remove $p$ again }
  $incr(overflow\text{-}line)$; $dvi\_goto(over\_col, overflow\text{-}line * thrice\text{-}x\text{-}height + 655360)$;
  $hbox(info[p], label\text{-}font, true)$;
  if $q \neq null$ then
    begin $hbox(equals\text{-}sign, label\text{-}font, true)$; $hbox(info[label\_for\_dot[q]], label\text{-}font, true)$;
    $hbox(plus\text{-}sign. label\_font, true)$; $dvi\_scaled((xx[p] - xx[q])/x\_ratio + (yy[p] - yy[q]) * fudge\text{-}factor)$:
    $dvi\text{-}out$ (", "); $dvi\_scaled((yy[q] - yy[p])/y\_ratio)$; $dvi\_out$(") ");
    end;
  $dvi\text{-}out(pop)$;
  end
```

This code is used in section 197.

200. ⟨ Adjust the maximum page width 200 ⟩ ≡

```
  if $overflow\text{-}line > 1$ then $page\text{-}width \leftarrow over\text{-}col + 10000000$;
        { overflow labels are estimated to occupy $10^7$ sp }
  if $page\text{-}width > max\_h$ then $max\text{-}h \leftarrow page\text{-}width$
```

This code is used in section 161.

201. 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.

202. 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 6 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 134 ⟩ +≡

```
for $k \leftarrow 0$ to 4095 do $b[k] \leftarrow 0$;
for $k \leftarrow font\_bc[gray\_font]$ to $font\_ec[gray\_font]$ do
  if $k \geq 1$ then
    if $k \leq 120$ then
      if $char\_exists(char\_info(gray\_font)(k))$ then
        begin $v \leftarrow c[k]$;
        repeat $b[v] \leftarrow k$; $v \leftarrow v + d[k]$;
        until $v > 4095$;
        end;
```

203. 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 $rho[O] = 2^{12}$.

⟨ Initialize global variables that depend on the font data 134 ⟩ +≡

```
for $j \leftarrow 0$ to 11 do
  begin $k \leftarrow two\_to\_the[j]$; $v \leftarrow k$;
  repeat $rho[v] \leftarrow k$; $v \leftarrow v + k + k$;
  until $v > 4095$;
  end;
$rho[O] \leftarrow 4096$;
```

204. ⟨ 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" }
```

205. 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 205 ⟩ ≡
  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 206⟩;
    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 207⟩;
    until b[a[j]] = 0;
    dvi_out (pop);
    end;
done:
```

This code is used in section 215.

206. ⟨ Move to column j in the DVI output 206⟩ ≡

```
  dvi-out(right4); dvi_four(round(unsc-x-ratio * j + unsc_slant_ratio * y) + delta-x)
```

This code is used in section 205.

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

```
⟨ Output the equivalent of k copies of character v 207 ⟩ ≡
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 205.

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

```
a: array [0 . . widest-row] of 0 . . 4095;  { bit patterns for twelve rows }
```

209. 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 }
```

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

```
⟨ Add more rows to a, until 12-bit entries are obtained 210⟩ ≡
  repeat ⟨Put the bits for the next row, times l, into a 211⟩;
    l ← l + l; decr(y);
  until l = 4096;
```

This code is used in section 215.

211. 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 211⟩ ≡
  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 212 ⟩;
      end;
```

This code is used in section 210.

```
212. define do-skip ≡ z ← 0; paint-black ← false
  define end-with(#) ≡
            begin #; cur-gf ← get-byte; goto donel; 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 212⟩ ≡
  loop begin continue: case cur-gf of
    sixty-four-cases (0): k ← cur-gf ;
    paint1 : k ← get-byte;
    paint2: k ← get-two-bytes;
    puint3: k ← get-three-bytes;
    eoc: goto donel ;
    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-O), 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-O; paint-black ← true);
    xxx1 , xxx2, xxx3, xxx4, yyy, no-op: begin skip_nop; goto continue;
      end:
    othercases bud-gf ( ´Improper␣opcode´)
    endcases;
    ⟨ Paint k bits and read another command 213⟩;
    end;
donel :
```

This code is used in section 211.

213. ⟨ Paint k bits and read another command 213⟩ ≡

```
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 212.

214. 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 214⟩ ≡

```
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 215.

215. We now have constructed the major components of the necessary routine: it simply remains to glue them all toget her 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 ← mux-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 210⟩;
    dvi_goto(0, delta-y − round( unsc_y_ratio * y)); ⟨ Typeset the pixels of the current row 205⟩;
    dvi-out (pop); ⟨ Advance to the next row that needs to be typeset; or return, if we're all done 214⟩;
    end;
exit : end:
```

216. 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 218 ⟩;
 cur-gf ← get-byte; init-str-ptr ← str-ptr;
 loop begin ⟨ Initialize variables for the next character 141 ⟩;
   while $(cur\text{-}gf \ge xxx1) \land (cur\text{-}gf \le no\text{-}op)$ do ⟨ Process a no-op command 151 ⟩;
   if $cur\text{-}gf = post$ then ⟨ Finish the DVI file and goto final-end 115 ⟩;
   if $cur\text{-}gf \ne boc$ then
     if $cur\text{-}gf \ne boc1$ then abort('Missing␣boc!');
   ⟨ Process a character 161 ⟩;
   cur-gf ← get-byte; str-ptr ← init_str_ptr; pool-ptr ← str_start[str-ptr];
   end;
final-end: end.
```

217. 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? }
```

218. 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 218 ⟩ ≡
  if $get\text{-}byte \ne pre$ then bad_gf('No␣preamble');
  if $get\text{-}byte \ne gf\_id\_byte$ 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 216.

219. 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.

220. **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.

The MFT processor

(Version 1.1, April 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.** This program converts a METAFONT source file to a TEX file. It was written by D. E. Knuth in June, 1985; a somewhat similar SAIL program had been developed in January, 1980.

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

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

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

Furthermore, '`%%%` ⟨ token_1 ⟩ . . . ⟨ token_n ⟩' introduces a change in MFT's formatting rules; all tokens after the first will henceforth be translated according to the current conventions for ⟨ token_1 ⟩. The tokens must be symbolic (i.e., not numeric or string tokens). For example, the input line

`%%% addto fill draw filldraw`

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

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

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

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

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

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

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

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

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

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

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

define *banner* ≡ `'This␣is␣MFT,␣Version␣1.1'`

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

If it is necessary to abort the job because of a fatal error, the program calls the '*jump-out*' procedure. which goes to the label *end-of-MFT*.

```
define end-of-MFT = 9999   { go here to wrap it up }
⟨ Compiler directives 4 ⟩
program MFT (mf_file, change-file. style-file, tex-file);
  label end-of-MFT;   { go here to finish}
  const ⟨ Constants in the outer block 8⟩
  type ⟨ Types in the outer block 12⟩
  var ⟨ Globals in the outer block 9⟩
    ⟨ Error handling procedures 29⟩
  procedure initialize;
    var ⟨ Local variables for initialization 14⟩
    begin ⟨ Set initial values 10 ⟩
    end:
```

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

```
⟨ Compiler directives 4⟩ ≡
  @{@&$C+, A+, D-43   { range check, catch arithmetic overflow, no debug overhead }
```

This code is used in section 3.

5. Labels are given symbolic names by the following definitions. We insert the label '*exit:*' just before the 'end' of a procedure in which we have used the 'return' statement defined below; the label '*restart*' is occasionally used at the very beginning of a procedure; and the label '*reswitch*' is occasionally used just prior to a case statement in which some cases change the conditions and we wish to branch to the newly applicable case. Loops that are set up with the loop construction defined below are commonly exited by going to '*done*' or to '*found*' or to '*not-found*', and they are sometimes repeated by going to '*continue*'.

```
define exit = 10   { go here to leave a procedure }
define restart = 20   { go here to start a procedure again }
define reswitch = 21   { go here to start a case statement again }
define continue = 22   { go here to resume a loop }
define done = 30   { go here to exit a loop}
define found = 31   { go here when you've found it }
define not-found = 32   { go here when you've found something else }
```

6. Here are some macros for common programming idioms.

```
define incr(#) ≡ # ← # + 1   { increase a variable by unity }
define decr(#) ≡ # ← # − 1   { decrease a variable by unity }
define loop ≡ while true do   { repeat over and over until a goto happens }
define do-nothing ≡   { empty statement }
define return ≡ goto exit   { terminate a procedure call }
format return ≡ nil
format loop ≡ xclause
```

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

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

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

```
define othercases ≡ others:   { default for cases not listed explicitly }
define endcases ≡ end   { follows the default case in an extended case statement }
format othercases ≡ else
format endcases ≡ end
```

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

```
⟨Constants in the outer block 8⟩ ≡
  max_bytes = 10000;  { the number of bytes in tokens; must be less than 65536 }
  max_names = 1000;  { number of tokens }
  hash_size = 353;  { should be prime }
  buf_size = 100;  { maximum length of input line }
  line_length = 80;  { lines of TEX output have at most this many characters, should be less than 256 }
```

This code is used in section 3.

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

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

define mark_harmless ≡ if history = spotless then history ← harmless_message
define mark_error ≡ history ← error_message
define mark_fatal ≡ history ← fatal_message
```

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

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

This code is used in section 3.

10.
```
⟨Set initial values 10⟩ ≡
  history ← spotless;
```

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

This code is used in section 3.

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

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

⟨ Types in the outer block 12 ⟩ ≡
ASCII-code = *0* . . 127; { seven-bit numbers }

See also sections 13, 50, and 52.

This code is used in section 3.

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

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

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

define *text-char* ≡ *char* { the data type of characters in text files }
define *first-text-char* = 0 { ordinal number of the smallest element of *text-char* }
define *last-text-char* = 127 { ordinal number of the largest, element of *text-char* }

⟨ Types in the outer block 12 ⟩ +≡
text-file = **packed file of** *text-char;*

14. ⟨ Local variables for initialization 14 ⟩ ≡
i: 0 . . last-text-char;

See also section 56.

This code is used in section 3.

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

⟨ Globals in the outer block 9 ⟩ +≡
xord: **array** *[text-char]* **of** *ASCII-code:* { specifies conversion of input characters }
xchr : **array** *[ASCII-code]* **of** *text-char;* { specifies conversion of output characters }

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

```
⟨ Set initial values 10⟩ +≡
  xchr['40] - '␣'; xchr['41] - '!'; xchr['42] - '"'; xchr['43] - '#'; xchr['44] ← '$';
  xchr['45] - '%'; xchr['46] - '&'; xchr['47] ← '''';
  xchr['50] ← '('; xchr['51] - ')'; xchr['52] + '*'; xchr['53] ← '+'; xchr['54] ← ',';
  xchr['55] + '-'; xchr['56] - '.'; xchr['57] ← '/';
  xchr ['60] - '0'; xchr['61] ← '1'; xchr['62] - '2'; xchr['63] - '3'; xchr['64] ← '4';
  xchr ['65] - '5'; xchr['66] ← '6'; xchr['67] - '7';
  xchr ['70] - '8'; xchr['71] ← '9'; xchr['72] ← ':'; xchr['73] - ';'; xchr['74] ← '<';
  xchr['75] - '='; xchr['76] + '>'; xchr['77] - '?';
  xchr['100] - '@'; xchr['101] - 'A'; xchr['102] - 'B'; xchr['103] + 'C'; xchr['104] ← 'D';
  xchr ['105] - 'E'; xchr['106] - 'F'; xchr['107] - 'G';
  xchr['110] + 'H'; xchr['111] - 'I'; xchr['112] - 'J'; xchr['113] - 'K'; xchr['114] ← 'L':
  xchr['115] - 'M'; xchr['116] - 'N'; xchr['117] - 'O';
  xchr ['120] - 'P'; xchr['121] ← 'Q'; xchr['122] ← 'R'; xchr['123] - 'S'; xchr['124] - 'T';
  xchr ['125]← 'U'; xchr['126] ← 'V'; xchr['127] - 'W';
  xchr ['130] - 'X'; xchr['131] ← 'Y'; xchr['132] - 'Z'; xchr['133] + '['; xchr['134] ← '\';
  xchr['135] - ']'; xchr['136] - '^'; xchr['137] - '_';
  xchr['140] - '`'; xchr['141] - 'a'; xchr['142] ← 'b'; xchr['143] ← 'c'; xchr['144] ← 'd';
  xchr['145] ← 'e'; xchr['146] ← 'f'; xchr['147] ← 'g';
  xchr['150] - 'h'; xchr['151] - 'i'; xchr['152] ← 'j'; xchr['153] - 'k'; xchr['154] - 'l';
  xchr['155] ← 'm'; xchr['156] + 'n'; xchr['157] ← 'o';
  xchr['160] - 'p'; xchr['161] ← 'q'; xchr['162] ← 'r'; xchr['163] - 's'; xchr['164] - 't':
  xchr['165] ← 'u'; xchr['166] - 'v'; xchr['167] ← 'w';
  xchr['170] - 'x'; xchr['171] ← 'y'; xchr['172] - 'z'; xchr['173] ← '{'; xchr['174] ← '|';
  xchr['175] ← '}'; xchr['176] - '~';
  xchr[0] - '␣'; xchr['177] ← '␣'; { ASCII codes 0 and '177 do not appear in text }
```

17. The ASCII code is "standard" only to a certain extent, since many computer installations have found it advantageous to have ready access to more than 94 printing characters. If MFT is being used on a garden-variety Pascal for which only standard ASCII codes will appear in the input and output files, it doesn't really matter what codes are specified in *xchr* [1 . . '37], but the safest policy is to blank everything out by using the code shown below.

However, other settings of *xchr* will make MFT more friendly on computers that have an extended character set, so that users can type things like '≠' instead of '<>', and so that MFT can echo the page breaks found in its input. People with extended character sets can assign codes arbitrarily, giving an *xchr* equivalent to whatever characters the users of MFT are allowed to have in their input files. Appropriate changes to MFT's *char-class* table should then be made. (Unlike TEX, each installation of METRFONT has a fixed assignment of category codes, called the *char-class* .) Such changes make portability of programs more difficult, so they should be introduced cautiously if at all.

```
⟨ Set initial values 10⟩ +≡
  for i ← 1 to '37 do xchr [i] - '␣';
```

18. The following system-independent code makes the *xord* array contain a suitable inverse to the information in *xchr*. Note that if $xchr[i] = xchr[j]$ where $i < j < '177$, the value of $xord[xchr[i]]$ will turn out to be j or more: hence, standard ASCII code numbers will be used instead of codes below $'40$ in case there is a coincidence.

⟨ Set initial values 10⟩ +≡

```
  for $i \leftarrow first\text{-}text\text{-}char$ to $lust\text{-}text\text{-}char$ do $xord[chr(i)] \leftarrow '177$;
  for $i \leftarrow 1$ to $'176$ do $xord[xchr[i]] \leftarrow i$
```

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

20. Terminal output is done by writing on file *term-out,* which is assumed to consist of characters of type *text-char:*

```
define prznt(#) ≡ write(term-out, #)   { 'print' means write on the terminal }
define print_ln(#) ≡ write-Zn(term-out, #)   { 'print' and then start new line }
define new-line ≡ write_ln(term-out)   { start new line on the terminal }
define print-d(#) ≡   { print information starting on a new line }
        begin new-line; print(#);
        end
```

⟨ Globals in the outer block 9 ⟩ +≡
term-out: text-file; { the terminal as an output file}

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

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

22. The *update-terminal* procedure is called when we want to make sure that everything we have output to the terminal so far has actually left the computer's internal buffers and been sent.

```
define update-terminal ≡ break(term-out)   { empty the terminal output buffer }
```

23. The main input comes from *mf-file;* this input may be overridden by changes in *chunge-file.* (If *change-file* is empty, there are no changes.) Furthermore the *style-file* is input first; it is unchangeable.

⟨ Globals in the outer block 9 ⟩ +≡
mf_file : *text-file* ; { primary input }
change-file : *text-file* ; { updates }
style-file: text-file; { formatting bootstrap }

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

```
procedure open-input;   { prepare to read the inputs }
  begin reset (mf_file); reset (change-file); reset (style-file);
  end:
```

25. The main output goes to *tex-file.*

⟨ Globals in the outer block 9 ⟩ +≡
tex-file : *text-file* ;

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

⟨ Set initial values 10⟩ +≡
  rewrite(tex-file);

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

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

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

```
function input-ln(var f : text-file): boolean;  { inputs a line or returns false }
  var final_limit: 0 . . buf-size;  { limit without trailing blanks }
  begin limit ← 0; final-limit ← 0;
  if eof (f) then input_ln ← false
  else begin while ¬eoln (f) do
      begin buffer [limit] ← xord[f↑]; get(f); incr(limit);
      if buffer[limit − 1] ≠ "␣" then final-limit ← limit;
      if limit = buf-size then
        begin while ¬eoln(f) do get(f);
        decr (limit);  { keep buffer [buf-size] empty }
        if final-limit > limit then final-limit ← limit;
        print_nl(´!␣Input␣line␣too␣long´); loc ← 0; error;
        end:
      end;
    rend-h (f); limit ← final-limit; input-h ← true;
    end;
  end;
```

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

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

```
define err-print (#) ≡
        begin new-line; print (#); error:
        end
```

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

See also section 31.

This code is used in section 3.

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

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

This code is used in section 29.

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

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

```
define fatal-error (#) ≡
        begin new-line; print (#); error; mark-fatal; jump-out;
        end
```

```
⟨ Error handling procedures 29⟩ +≡
procedure jump-out;
  begin goto end_of_MFT;
  end:
```

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

define *confusion*(#) ≡ *fatal_error*(`'!␣This␣can''t␣happen␣(',#,')'`)

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

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

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

```
⟨ Globals in the outer block 9 ⟩ +≡
line: integer;   { the number of the current line in the current file }
other-line: integer;   { the number of the current line in the input file that is not currently being read }
temp-line: integer;   { used when interchanging line with other-line }
limit: 0 . . buf-size;   { the last character position occupied in the buffer }
Zoc: 0 . . buf-size;   {the next character position to be read from the buffer }
input-has-ended: boolean;   { if true, there is no more input }
changing : boolean ;   { if true, the current line is from change-file }
styling: boolean;   { if true, the current line is from style-file }
```

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

```
  define change-changing ≡ changing ← ¬ changing ; temp-line ← other-line; other-line ← line;
        line + temp-line   { line ↔ other-line }
```

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

```
⟨ Globals in the outer block 9 ⟩ +≡
change-buffer: array [0 . . buf-size] of ASCII-code;
change-limit: 0 . . buf-size;   { the last position occupied in change-buffer }
```

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

```
function lines-dont-match: boolean;
  label exit;
  var k: 0 . . buf-size;   { index into the buffers }
  begin lines-dent-match + true;
  if change-limit ≠ limit then return;
  if limit > 0 then
    for k ← 0 to limit − 1 do
      if change-buffer [k] ≠ buffer [k] then return;
  lines-dont-match ← false;
exit: end;
```

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

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

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

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

This code is used in section 38.

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

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

This code is used in section 38.

41. ⟨ Move *buffer* and *limit* to *change_buffer* and *change_limit* 41 ⟩ ≡

```
  begin change_limit ← limit;
  if limit > 0 then
    for k ← 0 to limit − 1 do change_buffer[k] ← buffer[k];
  end
```

This code is used in sections 38 and 42.

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

```
procedure check-change ;  { switches to change_file if the buffers match }
  label exit;
  var n: integer;  { the number of discrepancies found }
    k: 0 . . buf-size;  { index into the buffers }
  begin if lines-dont-match then return;
  n ← 0;
  loop begin change-changing;  { now it's true }
    incr (line);
    if ¬ input-ln ( change-file) then
      begin err-print ( ´ !␣Change␣file␣ended␣before␣@y´); change-limit ← 0; change-changing;
          {false again }
      return;
      end;
    ⟨ If the current line starts with @y, report any discrepancies and return 43⟩;
    ⟨ Move buffer and limit to change-buffer and change-limit 41⟩;
    change-changing;  { now it's false }
    incr (line);
    if ¬input_ln (mf-file) then
      begin err-print (´!␣MF␣file␣ended␣during␣a␣change´); input-has-ended + true; return;
      end;
    if lines-dont-match then incr(n);
    end;
exit : end;
```

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

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

This code is used in section 42.

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

```
⟨ Initialize the input system 44 ⟩ ≡
  begin open-input; line ← 0; other-line ← 0;
  changing ← true: prime-the-change-buffer; change-changing;
  styling ← true: limit ← 0; loc ← 1: buffer[0] ← "␣"; input-has-ended ← false;
  end
```

This code is used in section 112.

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

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

```
46. ⟨ Read from mf-file and maybe turn on changing 46 ⟩ ≡
  begin incr(line);
  if ¬input_ln(mf-file) then input-has-ended ← true
  else if limit = change-limit then
      if buffer[0] = change-buffer[0] then
        if change-limit > 0 then check-change;
  end
```

This code is used in section 45.

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

This code is used in section 45.

48. ⟨ Read from *change-file* and-maybe turn *off changing* 48 ⟩ ≡

```
begin incr (line);
if ¬input_ln (change-file) then
  begin err-print (´! ␣Change␣f ile␣ended␣without␣@z´); buffer[0] ← "@"; buffer[1] ← "z"; limit ← 2;
  end;
if limit > 1 then   { check if the change has ended }
  if buffer[0] = "@" then
    begin if (buffer[1] ≥ "X") A (buffer[1] ≤ "Z") then buffer[1] ← buffer[1] + "z" − "Z";
          { lowercasify }
    if (buffer[1] = "x")|(buffer[1] = "y") then
      begin loc ← 2; err-print (´!␣Where␣is␣the␣matching␣@z?´);
      end
    else if buffer [ 1] = "z" then
        begin prime-the-change-buffer; change-changing;
        end;
    end;
end
```

This code is used in section 45.

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

⟨ Check that all changes have been read 49 ⟩ ≡

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

This code is used in section 112.

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

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

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

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

⟨ Globals in the outer block 9 ⟩ +≡
byte-mem: packed array *[0 . . max-bytes]* of *ASCII_code*; { characters of names}
byte-start: array [0 . . *maxnames]* of *sixteen-bits;* { directory into *byte-mem* }
link: array [0 . . *maxnames]* of *sixteen-bits;* { hash table links }
ilk: array [0 . . *maxnames]* of *sixteen-bits;* { type codes }

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

A '*name-pointer*' variable, which signifies a name, is an index into *byte-start.* The actual sequence of characters in the name pointed to by *p* appears in positions *byte_start*[*p*] to *byte_start*[*p* + 1] − 1, inclusive, of *byte-mem* .

We usually have *byte-start[name-ptr]* = *byte-ptr,* which is the starting position for the next name to be stored in *byte-mem*.

define *length(#)* ≡ *byte_start*[# + 1] − *byte_start*[#] { the length of a name}

⟨Types in the outer block 12⟩ +≡
name-pointer = *0* . . *max-names;* { identifies a name }

53. ⟨ Globals in the outer block 9⟩ +≡
name-ptr: name-pointer; { first unused position in *byte-start* }
byte-ptr: 0 . . *max-bytes;* { first unused position in *byte-mem* }

54. ⟨ Set initial values 10⟩ +≡
byte-start [0] ← *0; byte-ptr* ← *0; byte_start*[1] ← *0;* { this makes name 0 of length zero }
name-ptr ← 1;

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

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

⟨ Globals in the outer block 9 ⟩ +≡
id-first: 0 . . *buf_size*; { where the current token begins in the buffer }
id-lot: 0 . . *buf_size*; {just after the current token in the buffer }
hash: array [0 . . *hash-size]* of *sixteen-bits;* { heads of hash lists }

56. Initially all the hash lists are empty.

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

57. (Set initial values 10) $+\equiv$

```
for $h \leftarrow 0$ to $hash\text{-}size - 1$ do $hash[h] \leftarrow 0$:
```

58. Here now is the main procedure for finding tokens.

```
function lookup: name-pointer:   { finds current token}
  label found;
  var i: 0 . . buf-size;   { index into buffer }
    h: 0 . . hash-size;   { hash code }
    k: 0 . . max_bytes;   { index into byte-mem }
    1: 0 . . buf-size;   { length of the given token }
    p: name-pointer;   { where the token is being sought }
  begin $l \leftarrow id\_loc - id\text{-}first$;   { compute the length }
  ( Compute the hash code h 59⟩;
  ( Compute the name location p 60 ⟩;
  if $p = name\text{-}ptr$ then ( Enter a new name into the table at position p 62);
  lookup $\leftarrow p$;
  end;
```

59. A simple hash code is used: If the sequence of ASCII codes is $c_1c_2 \ldots$ c,. its hash value will be

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

```
( Compute the hash code h 59⟩ ≡
  $h \leftarrow buffer[id\_first]$; $i \leftarrow id\text{-}first + 1$;
  while $i < id\text{-}lot$ do
    begin $h \leftarrow (h + h + buffer[i]) \bmod hash\text{-}size$; incr (i);
    end
```

This code is used in section 58.

60. If the token is new, it will be placed in position $p = name\text{-}ptr$, otherwise p will point to its existing location.

```
⟨ Compute the name location p 60 ) ≡
  $p \leftarrow hash[h]$;
  while $p \neq 0$ do
    begin if $length(p) = l$ then (Compare name p with current token, goto found if equal 61);
    $p \leftarrow link[p]$;
    end;
  $p \leftarrow$ name-ptr;   { the current token is new }
  $link[p] \leftarrow hash[h]$; $hash[h] \leftarrow p$;   { insert p at beginning of hash list }
found:
```

This code is used in section 58.

61. (Compare name p with current token, **goto** *found* if equal 61) $\equiv$

```
  begin $i \leftarrow id\text{-}first$; $k \leftarrow byte\text{-}start[p]$;
  while $(i < id\_loc) \wedge (buffer[i] = byte\text{-}mem[k])$ do
    begin incr (i); incr (k);
    end:
  if $i = id\text{-}lot$ then goto found;   { all characters agree }
  end
```

This code is used in section 60.

62. When we begin the following segment of the program. $p = name\text{-}ptr$.

```
⟨ Enter a new name into the table at position p 62 ⟩ ≡
  begin if byte-ptr + l > max-bytes then overflow('byte␣memory');
  if name-ptr + 1 > max-names then overflow ( 'name');
  i ← id_first;  { get ready to move the token into byte-rnem }
  while i < id-Zoc do
    begin byte-mem [byte_ptr] ← buffer[i]; incr(byte-ptr); incr(i);
    end;
  incr( name-ptr); byte-start [name-ptr] ← byte-ptr; ⟨Assign the default value to ilk[p] 63⟩;
  end
```

This code is used in section 58.

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

define *indentation* = 0 { internal code for space at beginning of a line }
define *end-of-line* = 1 { internal code for hypothetical token at end of a line }
define *end-of-file* = 2 { internal code for hypothetical token at end of the input }
define *verbatim* = 3 { internal code for the token '%%' }
define *set-format* = 4 { internal code for the token '%%%' }
define *mft_comment* = 5 { internal code for the token '%%%%' }
define *min-action-type* = 6 {smallest code for tokens that produce "real" output }
define *numeric-token* = 6 { internal code for tokens like '3.14159' }
define *string-token* = 7 { internal code for tokens like '"pie"'}
define *min-symbolic-token* = 8 { smallest internal code for a symbolic token }
define *op* = 8 { internal code for tokens like '**sqrt**' }
define *command* = 9 { internal code for tokens like '**addto**' }
define *endit* = 10 { internal code for tokens like '**f** i' }
define *binary* = 11 { internal code for tokens like 'and' }
define *abinary* = 12 { internal code for tokens like '+' }
define *bbinary* = 13 { internal code for tokens like 'step' }
define *ampersand* = 14 { internal code for the token '&' }
define *pyth-sub* = 15 { internal code for the token '+-+' }
define us-is = 16 { internal code for tokens like ']' }
define *bold* = 17 { internal code for tokens like '**nullpen**' }
define *type-name* = 18 { internal code for tokens like 'numeric' }
define *path-join* = 19 { internal code for the token '..' }
define *colon* = 20 { internal code for the token ':' }
define *semicolon* = 21 { internal code for the token ';' }
define *backslash* = 22 { internal code for the token '\' }
define *double-back* = 23 { internal code for the token '\\' }
define *less-or-equal* = 24 { internal code for the token '<=' }
define *greater-or-equal* = 25 { internal code for the token '>=' }
define *not-equal* = 26 { internal code for the token '<>' }
define *sharp* = 27 { internal code for the token '#' }
define *comment* = 28 { internal code for the token '%' }
define *recomment* = 29 { internal code used to resume a comment after 'I . . . I' }
define *min_suffix* = 30 { smallest code for symbolic tokens in suffixes }
define *internal* = 30 { internal code for tokens like 'pausing' }
define *input-command* = 31 { internal code for tokens like 'input'}
define *special-tag* = 32 { internal code for tags that take at most one subscript }
define *tag* = 33 { internal code for nonprimitive tokens }

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

This code is used in section 62.

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

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

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

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

⟨ Store all the primitives 65 ⟩ ≡

```
  id-lot ← 18;
  pr2(".")(".")(path_join);
  prl("[")(as-is);
  pr1("]")(as-is);
  prl("}")(us-is);
  prl("{")(us-is);
  prl(":")(colon);
  pr2(":")("=")(us-is);
  prl(",")(us-is);
  prl(";")(semicolon);
  prl("\")(backslash);
  pr2("\")("\")(double-buck);
  pr5("a")("d")("d")("t")("o")(command);
  pr2("a")("t")(bbinary);
  pr7("a")("t")("l")("e")("a")("s")("t")(op);
  pr10("b")("e")("g")("i")("n")("g")("r")("o")("u")("p")(command);
  pr8("c")("o")("n")("t")("r")("o")("l")("s")(op);
  pr4("c")("u")("l")("l")(command);
  pr4("c")("u")("r")("l")(op);
  pr10("d")("e")("l")("i")("m")("i")("t")("e")("r")("s")(command);
  pr7("d")("i")("s")("p")("l")("a")("y")(command);
  pr8("e")("n")("d")("g")("r")("o")("u")("p")(endit);
  pr8("e")("v")("e")("r")("y")("j")("o")("b")(command);
  pr6("e")("x")("i")("t")("i")("f")(command);
  prll ("e")("x")("p")("a")("n")("d")("a")("f")("t")("e")("r")(command);
  pr4("f")("r")("o")("m")(bbinury);
  pr8("i")("n")("w")("i")("n")("d")("o")("w")(bbinary);
  pr7("i")("n")("t")("e")("r")("i")("m")(command);
  pr3("l")("e")("t")(command);
  prll ("n")("e")("w")("i")("n")("t")("e")("r")("n")("a")("l")(command);
  pr2("o")("f")(command);
  pr10("o")("p")("e")("n")("w")("i")("n")("d")("o")("w")(command);
  pr10("r")("a")("n")("d")("o")("m")("s")("e")("e")("d")(command);
  pr4("s")("a")("v")("e")(command);
  pr10("s")("c")("a")("n")("t")("o")("k")("e")("n")("s")(command):
  pr7("s")("h")("i")("p")("o")("u")("t")(command):
  pr4("s")("t")("e")("p")(bbinary);
  pr3("s")("t")("r")(command);
  pr7("t")("e")("n")("s")("i")("o")("n")(op);
  pr2("t")("o")(bbinary);
  pr5("u")("n")("t")("i")("l")(bbinary);
  pr3("d")("e")("f")(command);
  pr6("v")("a")("r")("d")("e")("f")(command);
  pr10("p")("r")("i")("m")("a")("r")("y")("d")("e")("f")(command);
  pr12("s")("e")("c")("o")("n")("d")("a")("r")("y")("d")("e")("f")(command);
  pr11("t")("e")("r")("t")("i")("a")("r")("y")("d")("e")("f")(command);
```

See also sections 66, 67, 68, 69, 70, and 71

This code is used in section 112.

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

⟨ Store all the primitives 65⟩ +≡
pr6("e")("n")("d")("d")("e")("f")(*endit*);
pr3("f")("o")("r")(*command*);
pr11("f")("o")("r")("s")("u")("f")("f")("i")("x")("e")("s")(*command*);
pr7("f")("o")("r")("e")("v")("e")("r")(*command*);
pr6("e")("n")("d")("f")("o")("r")(*endit*);
pr5("q")("u")("o")("t")("e")(*command*);
pr4("e")("x")("p")("r")(*command*);
pr6("s")("u")("f")("f")("i")("x")(*command*);
pr4("t")("e")("x")("t")(*command*);
pr7("p")("r")("i")("m")("a")("r")("y")(*command*);
pr9("s")("e")("c")("o")("n")("d")("a")("r")("y")(*command*);
pr8("t")("e")("r")("t")("i")("a")("r")("y")(*command*);
pr5("i")("n")("p")("u")("t")(*input_command*);
pr8("e")("n")("d")("i")("n")("p")("u")("t")(*bold*);
pr2("i")("f")(*command*);
pr2("f")("i")(*endit*);
pr4("e")("l")("s")("e")(*command*);
pr6("e")("l")("s")("e")("i")("f")(*command*);
pr4("t")("r")("u")("e")(*bold*);
pr5("f")("a")("l")("s")("e")(*bold*);
pr11("n")("u")("l")("l")("p")("i")("c")("t")("u")("r")("e")(*bold*);
pr7("n")("u")("l")("l")("p")("e")("n")(*bold*);
pr7("j")("o")("b")("n")("a")("m")("e")(*bold*);
pr10("r")("e")("a")("d")("s")("t")("r")("i")("n")("g")(*bold*);
pr9("p")("e")("n")("c")("i")("r")("c")("l")("e")(*bold*);
pr4("g")("o")("o")("d")(*special_tag*);

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

⟨ Store all the primitives 65⟩ +≡

```
pr13("n")("o")("r")("m")("a")("l")("d")("e")("v")("i")("a")("t")("e")(op);
pr3("o")("d")("d")(op);
pr5("k")("n")("o")("w")("n")(op);
pr7("u")("n")("k")("n")("o")("w")("n")(op);
pr3("n")("o")("t")(op);
pr7("d")("e")("c")("i")("m")("a")("l")(op);
pr7("r")("e")("v")("e")("r")("s")("e")(op);
pr8("m")("a")("k")("e")("p")("a")("t")("h")(op);
pr7("m")("a")("k")("e")("p")("e")("n")(op);
pr11("t")("o")("t")("a")("l")("w")("e")("i")("g")("h")("t")(op);
pr3("o")("c")("t")(op);
pr3("h")("e")("x")(op);
pr5("A")("S")("C")("I")("I")(op);
pr4("c")("h")("a")("r")(op);
pr6("l")("e")("n")("g")("t")("h")(op);
pr13("t")("u")("r")("n")("i")("n")("g")("n")("u")("m")("b")("e")("r")(op);
pr5("x")("p")("a")("r")("t")(op);
pr5("y")("p")("a")("r")("t")(op);
pr6("x")("x")("p")("a")("r")("t")(op);
pr6("x")("y")("p")("a")("r")("t")(op);
pr6("y")("x")("p")("a")("r")("t")(op);
pr6("y")("y")("p")("a")("r")("t")(op);
pr4("s")("q")("r")("t")(op);
pr4("m")("e")("x")("p")(op);
pr4("m")("l")("o")("g")(op);
pr4("s")("i")("n")("d")(op);
pr4("c")("o")("s")("d")(op);
pr5("f")("l")("o")("o")("r")(op);
pr14("u")("n")("i")("f")("o")("r")("m")("d")("e")("v")("i")("a")("t")("e")(op);
pr10("c")("h")("a")("r")("e")("x")("i")("s")("t")("s")(op);
pr5("a")("n")("g")("l")("e")(op);
pr5("c")("y")("c")("l")("e")(op);
```

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

⟨ Store all the primitives 65⟩ +≡

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

69. ⟨Store all the primitives 65⟩ +≡

```
pr4("y")("e")("a")("r")(internal);
pr5("m")("o")("n")("t")("h")(internal);
pr3("d")("a")("y")(internal);
pr4("t")("i")("m")("e")(internal);
pr8("c")("h")("a")("r")("c")("o")("d")("e")(internal);
pr7("c")("h")("a")("r")("f")("a")("m")(internal);
pr6("c")("h")("a")("r")("w")("d")(internal);
pr6("c")("h")("a")("r")("h")("t")(internal);
pr6("c")("h")("a")("r")("d")("p")(internal);
pr6("c")("h")("a")("r")("i")("c")(internal);
pr6("c")("h")("a")("r")("d")("x")(internal);
pr6("c")("h")("a")("r")("d")("y")(internal);
pr10("d")("e")("s")("i")("g")("n")("s")("i")("z")("e")(internal);
pr4("h")("p")("p")("p")(internal);
pr4("v")("p")("p")("p")(internal);
pr7("x")("o")("f")("f")("s")("e")("t")(internal);
pr7("y")("o")("f")("f")("s")("e")("t")(internal);
pr7("p")("a")("u")("s")("i")("n")("g")(internal);
pr12("s")("h")("o")("w")("s")("t")("o")("p")("p")("i")("n")("g")(internal);
pr10("f")("o")("n")("t")("m")("a")("k")("i")("n")("g")(internal);
pr8("p")("r")("o")("o")("f")("i")("n")("g")(internal);
pr9("s")("m")("o")("o")("t")("h")("i")("n")("g")(internal);
pr12("a")("u")("t")("o")("r")("o")("u")("n")("d")("i")("n")("g")(internal);
pr11("g")("r")("a")("n")("u")("l")("a")("r")("i")("t")("y")(internal);
pr6("f")("i")("l")("l")("i")("n")(internal);
pr12("t")("u")("r")("n")("i")("n")("g")("c")("h")("e")("c")("k")(internal);
pr12("w")("a")("r")("n")("i")("n")("g")("c")("h")("e")("c")("k")(internal);
```

70. Still more.

⟨Store all the primitives 65⟩ +≡

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

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

```
⟨ Store all the primitives 65 ⟩ +≡
  pr3("e")("n")("d")(endit);
  pr4("d")("u")("m")("p")(endit);
  pr9("b")("a")("t")("c")("h")("m")("o")("d")("e")(bold);
  pr11("n")("o")("n")("s")("t")("o")("p")("m")("o")("d")("e")(bold);
  pr10("s")("c")("r")("o")("l")("l")("m")("o")("d")("e")(bold);
  pr13("e")("r")("r")("o")("r")("s")("t")("o")("p")("m")("o")("d")("e")(bold);
  pr5("i")("n")("n")("e")("r")(command);
  pr5("o")("u")("t")("e")("r")(command);
  pr9("s")("h")("o")("w")("t")("o")("k")("e")("n")(command);
  pr9("s")("h")("o")("w")("s")("t")("a")("t")("s")(bold);
  pr4("s")("h")("o")("w")(command);
  pr12("s")("h")("o")("w")("v")("a")("r")("i")("a")("b")("l")("e")(command);
  pr16("s")("h")("o")("w")("d")("e")("p")("e")("n")("d")("e")("n")("c")("i")("e")("s")(bold);
  pr7("c")("o")("n")("t")("o")("u")("r")(command);
  pr10("d")("o")("u")("b")("l")("e")("p")("a")("t")("h")(command);
  pr4("a")("l")("s")("o")(command);
  pr7("w")("i")("t")("h")("p")("e")("n")(command);
  pr10("w")("i")("t")("h")("w")("e")("i")("g")("h")("t")(command);
  pr8("d")("r")("o")("p")("p")("i")("n")("g")(command);
  pr7("k")("e")("e")("p")("i")("n")("g")(command);
  pr7("m")("e")("s")("s")("a")("g")("e")(command);
  pr10("e")("r")("r")("m")("e")("s")("s")("a")("g")("e")(command);
  pr7("e")("r")("r")("h")("e")("l")("p")(command);
  pr8("c")("h")("a")("r")("l")("i")("s")("t")(command);
  pr8("l")("i")("g")("t")("a")("b")("l")("e")(command);
  pr10("e")("x")("t")("e")("n")("s")("i")("b")("l")("e")(command);
  pr10("h")("e")("a")("d")("e")("r")("b")("y")("t")("e")(command);
  pr9("f")("o")("n")("t")("d")("i")("m")("e")("n")(command);
  pr2("=")(":")(as_is); pr4("k")("e")("r")("n")(binary);
  pr7("s")("p")("e")("c")("i")("a")("l")(command);
  pr10("n")("u")("m")("s")("p")("e")("c")("i")("a")("l")(command);
  pr1("%")(comment);
  pr2("%")("%")(verbatim);
  pr3("%")("%")("%")(set_format);
  pr4("%")("%")("%")("%")(mft_comment);
  pr1("#")(sharp);
```

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

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

```
⟨Globals in the outer block 9⟩ +≡
translation: array [ASCII_code] of name_pointer;
i: ASCII_code;  {index into translation}
```

73.

```
⟨Store all the translations 73⟩ ≡
  for i ← 0 to 127 do translation[i] ← 0;
  tr2("\")("$"); translation["$"] ← cur_tok;
  tr2("\")("#"); translation["#"] ← cur_tok;
  tr2("\")("&"); translation["&"] ← cur_tok;
  tr2("\")("{"); translation["{"] ← cur_tok;
  tr2("\")("}"); translation["}"] ← cur_tok;
  tr2("\")("_"); translation["_"] ← cur_tok;
  tr2("\")("%"); translation["%"] ← cur_tok;
  tr4("\")("B")("S")("␣"); translation["\"] ← cur_tok;
  tr4("\")("H")("A")("␣"); translation["^"] ← cur_tok;
  tr4("\")("T")("I")("␣"); translation["~"] ← cur_tok;
  tr5("\")("a")("s")("t")("␣"); translation["*"] ← cur_tok;
  tr4("\")("A")("M")("␣"); tr_amp ← cur_tok;
  tr4("\")("B")("L")("␣"); tr_skip ← cur_tok;
  tr4("\")("S")("H")("␣"); tr_sharp ← cur_tok;
  tr4("\")("P")("S")("␣"); tr_ps ← cur_tok;
  tr4("\")("l")("e")("␣"); tr_le ← cur_tok;
  tr4("\")("g")("e")("␣"); tr_ge ← cur_tok;
  tr4("\")("n")("e")("␣"); tr_ne ← cur_tok;
  tr5("\")("q")("u")("a")("d"); tr_quad ← cur_tok;
```

This code is used in section 112.

74.

```
⟨Globals in the outer block 9⟩ +≡
tr_le, tr_ge, tr_ne, tr_amp, tr_sharp, tr_skip, tr_ps, tr_quad: name_pointer;  {special translations}
```

75. **Inputting the next token.** MFT's lexical scanning routine is called *get-next.* This procedure inputs the next token of METAFONT input and puts its encoded meaning into two global variables, *cur-type* and *cur-tok*.

⟨ Globals in the outer block 9 ⟩ +≡
cur-type: eight-bits; { type of token just scanned }
cur-tok: integer; { hash table or buffer location }
prev-type: eight-bits; { previous value of *cur-type* }
prev-tok: integer; { previous value of *cur-tok* }

76. ⟨ Set initial values 10 ⟩ +≡
cur-type ← *end-of-line; cur-tok* ← *0;*

77. Two global state variables affect the behavior of *get-next:* A space will be considered significant when *start-of-line* is *true,* and the buffer will be considered devoid of information when *empty-buffer* is *true.*

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

78. The 128 *ASCII-code* characters are grouped into classes by means of the *char-class* table. Individual class numbers have no semantic or syntactic significance, expect in a few instances defined here. There's also *max-class.* which can be used as a basis for additional class numbers in nonstandard extensions of METRFONT.

define *digit-class* = 0 { the class number of 0123456789)
define *period-class* = 1 { the class number of '.' }
define *space-class* = 2 { the class number of spaces and nonstandard characters }
define *percent-class* = 3 { the class number of '%' }
define *string_class* = 4 { the class number of '"' }
define *right-paren-class* = 8 { the class number of ')' }
define *isolated-classes* ≡ 5, 6, 7, 8 { characters that make length-one tokens only }
define *letter-class* = 9 { letters and the underline character}
define *left-bracket-class* = 17 { '[' }
define *right-bracket-class* = 18 { ']' }
define *invalid-class* = 20 { bad character in the input }
define *end-line-class* = 21 { end of an input line (MFT only) }
define *max-class* = 21 { the largest class number }

⟨ Globals in the outer block 9 ⟩ +≡
char-class: array *[ASCII-code]* of 0 . . *max-class;* { the class numbers }

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

```
  define carriage-return = '15   {special code placed in buffer[limit] }
⟨ Set initial values 10⟩ +≡
  for i ← "0" to "9" do char_class[i] ← digit-class;
  char-class ["."] ← period_class; char-class ["␣"] ← space-class: char-class ["%"] ← percent-class :
  char-class [""""] ← string-class ;
  char-class [","] ← 5; char-class [";"] ← 6; char-class ["("] ← 7; char_class [")"] ← right-paren-class ;
  for i ← "A" to "Z" do char_class[i] ← letter-class;
  for i ← "a" to "z" do char_class[i] ← letter-class;
  char-class ["_"] ← letter-class;
  char-class["<"] ← 10; char_class["="] ← 10; char-class [">"] ← 10; char_class[":"] ← 10:
  char-class ["I"] ← 10;
  char_class["`"] ← 11; char_class["´"] ← 11;
  char-class["+"] ← 12; char_class["-"] ← 12;
  char_class["/"] ← 13; char-class ["*"] ← 13; char_class["\"] ← 13;
  char-class ["!"] ← 14; char-class ["?"] ← 14;
  char-class ["#"] ← 15; char_class["&"] ← 15; char_class["@"] ← 15; char_class["$"] ← 15;
  char_class["^"] ← 16; char_class["~"] ← 16;
  char-class ["["] ← left-bracket-class; char_class["]"] ← right-bracket-class;
  char_class["{"] ← 19; char_class["}"] ← 19;
  for i ← 0 to "␣" − 1 do char_class[i] ← invalid-class;
  char-class [carriage-return] ← end-line-class;
  char_class[127] ← invalid-class ;
```

80. And now we're ready to take the plunge into *get-next* itself.

```
  define switch = 25   { a label in get-next }
  define pass-digits = 85   { another }
  define pass-fraction = 86   { and still another, although goto is considered harmful}
procedure get-next;   { sets cur-type and cur-tok to next token }
  label switch, pass-digits, pass-fraction. done, found, exit ;
  var c: ASCII-code;   { the current character in the buffer }
    class: ASCII-code ;   { its class number }
  begin prev-type ← cur-type; prev-tok ← cur-tok;
  if empty-buffer then ⟨Bring in a new line of input; return if the file has ended 85⟩;
switch: c ← buffer[loc]; id-first ← loc; incr (loc); class ← char-class [c]; ⟨ Branch on the class, scan the
        token; return directly if the token is special, or goto found if it needs to be looked up 81⟩;
found: id_loc ← loc; cur-tok ← lookup; cur-type ← ilk[cur_tok];
exit: end:
```

```
81. define emit(#) ≡ begin cur-type ← #; cur-tok ← id-first; return: end
⟨ Branch on the class, scan the token; return directly if the token is special, or goto found if it needs to
      be looked up 81⟩ ≡
  case class of
  digit-class: goto pass-digits;
  period-class: begin class ← char-class [buffer [loc]];
    if class > period-class then goto switch   { ignore isolated '.' }
    else if class < period-class then goto pass-fraction;   { class = digit-class }
    end;
  space-class: if start-of-line then emit (indentation)
    else goto switch;
  end-line-class: emit (end-of-line);
  string-class: ⟨ Get a string token and return 82⟩;
  isolated-classes: goto found;
  invalid-class: ⟨ Decry the invalid character and goto switch 84⟩;
  othercases do-nothing   { letters, etc. }
  endcases;
  while char-class [ buffer [loc]] = class do incr (Zoc);
  goto found;
pass-digits: while char-class [ buffer [ loc]] = digit-class do incr (Zoc);
  if buffer [Eoc] ≠ " . " then goto done;
  if char_class[buffer[loc + 1]] ≠ digit-class then goto done;
  incr(loc);
pass-fraction: repeat incr (loc);
  until char-class [ buffer [loc]] ≠ digit-class;
done: emit (numeric-token)
```

This code is used in section 80.

```
82. ⟨ Get a string token and return 82⟩ ≡
  loop begin if buffer[loc] = """"" then
      begin incr (Zoc); emit (string-token);
      end;
    if loc = limit then ⟨ Decry the missing string delimiter and goto switch 83⟩;
    incr(loc);
    end
```

This code is used in section 81.

```
83. ⟨ Decry the missing string delimiter and goto switch 83⟩ ≡
  begin err-print ( ´ !␣Incomplete␣string␣will␣be␣ignored´); goto switch;
  end
```

This code is used in section 82.

```
84. ⟨ Decry the invalid character and goto switch 84⟩ ≡
  begin err-print ( ´!␣Invalid␣character␣will␣be␣ignored´); goto switch;
  end
```

This code is used in section 81.

```
85. ⟨ Bring in a new line of input; return if the file has ended 85⟩ ≡
  begin get-line:
  if input-has-ended then emit (end-of-file);
  buffer[limit] ← carriage-return: loc ← 0: start-of-line ← true; empty_buffer ← false;
  end
```

This code is used in section 80.

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

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

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

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

88. MFT calls *flush_buffer*(*out_ptr*, *false*) before it has input anything. We initialize the output variables so that the first line of the output file will be 'input mftmac'.

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

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

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

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

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

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

```
procedure break-out; { finds a way to break the output line }
  label exit;
  var k: 0 . . line-length; { index into out-buf }
    c, d: ASCII-code; { characters from the buffer }
  begin k ← out-ptr;
  loop begin if k = 0 then ⟨ Print warning message, break the line, return 92⟩;
    d ← out-buf [k];
    if d = "␣" then
      begin flush_buffer (k, false); return;
      end;
    if (d = "\") ∧ (out-buf [k − 1] ≠ "\") then   { in this case k > 1 }
      begin flush_buffer(k − 1, true); return;
      end;
    decr(k);
    end;
exit: end:
```

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

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

This code is used in section 91.

93. To output a string of bytes from *byte-mem*, we call *out-str*.

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

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

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

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

```
procedure out-mat-and-name (n : ASCII-code; p : name-pointer);
  begin out ("\"); out(n);
  if length(p) = 1 then out-name(p)
  else begin out ("{"); out-name(p); out ("}");
    end;
  end;
```

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

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

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

```
procedure do-the-translation;
  label restart, reswitch, done, exit;
  var k: 0 . . buf_size;  { looks ahead in the buffer }
    t: integer;  { type that spreads to new tokens}
  begin restart: if out-ptr > 0 then flush-buffer(out-ptr, false);
  empty-buffer ← true;
  loop begin get-next ;
    if start-of-line then ⟨ Do special actions at the start of a line 98⟩;
  reswitch: case cur-type of
    numeric-token: ⟨ Translate a numeric token or a fraction 105⟩;
    string-token: ⟨ Translate a string token 99⟩;
    indentation: out-str (tr_quad);
    end-of-line, mft-comment: ⟨Wind up a line of translation and goto restart, or finish a I . . . I segment
          and goto reswitch 110⟩;
    end-of-file : return;
    ⟨ Cases that translate primitive tokens 100⟩
    comment, recomment: ⟨ Translate a comment and goto restart, unless there's a I . . . I segment 108⟩;
    verbatim: ⟨ Copy the rest of the current input line to the output, then goto restart 109⟩;
    set-format: ⟨ Change the translation format of tokens. and goto restart or reswitch 111⟩;
    internal, special-tag, tag: ⟨ Translate a tag and possible subscript 106⟩;
    end;  { all cases have been listed }
    end;
exit: end:
```

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

```
  if cur-type ≥ min_action_type then
    begin out ("$"); start-of-line ← false;
    case cur-type of
    endit: out2 ("\")("!");
    binary, abinary, bbinary, ampersand,pyth-sub: out2("{")("}");
    othercases do-nothing
    endcases;
    end
  else if cur- type = end-of-line then
      begin out-str ( tr-skip); goto restart;
      end
    else if cur-type = mft-comment then goto restart
```

This code is used in section 97.

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

⟨ Translate a string token 99⟩ ≡

```
  begin out2 ("\")( "7"); copy( cur-tok);
  end
```

This code is used in section 97.

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

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

See also sections 101, 102, and 103.

This code is used in section 97.

101. Here are a few more easy cases.

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

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

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

103. The remaining primitive is slightly special.

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

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

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

This code is used in section 103.

```
105. ⟨Translate a numeric token or a fraction 105⟩ ≡
  if buffer[loc] = "/" then
    if char-class[buffer[loc + 1]] = digit-class then   { it's a fraction }
      begin out5("\")("f")("r")("a")("c"); copy(cur_tok); get-next; out2("/")("{"); get-next;
      copy(cur-tok); out("}");
      end
    else copy(cur-tok)
  else copy(cur-tok)
```

This code is used in section 97.

```
106. ⟨Translate a tag and possible subscript 106⟩ ≡
  begin if length(cur-tok) = 1 then out-name(cur-tok)
  else out-mat-and-name("\", cur-tok);
  get-next;
  if byte-mem[byte-start[prev_tok]] = "´" then goto reswitch;
  case prev-type of
  internal: begin if (cur-type = numeric-token) ∨ (cur-type ≥ min_suffix) then out2("\")(",");
    goto reswitch;
    end;
  special-tag: if cur-type < min_suffix then goto reswitch
    else begin out("."); cur-type ← internal; goto reswitch;
      end;
  tag: begin if cur-type = tag then
      if byte-mem[byte-start[cur-tok]] = "´" then goto reswitch;
            { a sequence of primes goes on the main line }
    if (cur-type = numeric-token) ∨ (cur-type ≥ min_suffix) then ⟨Translate a subscript 107⟩
    else if cur-type = sharp then out-str(tr_sharp)
      else goto reswitch;
    end;
  end;   { there are no other cases }
  end
```

This code is used in section 97.

107. ⟨ Translate a subscript 107⟩ ≡

```
  begin out2 ("_")( "{");
  loop begin if cur-type ≥ min_suffix then out_name( cur_tok)
    else copy( cur-tok);
    if prev-type = special-tag then
      begin get_next : goto done:
      end:
    get-next;
    if cur_type < min_suffix then
      if cur-type ≠ numeric-token then goto done:
    if cur-type = prev-type then
      if cur-type = numeric-token then out2 ("\")(",")
      else if char_class[byte_mem[byte_start[cur_tok]]] = char_class[byte_mem[byte_start[prev_tok]]] then
           if byte-mem [byte-start [prev_tok]] ≠ " . " then out (". ")
           else out2 ("\")(", ");
    end;
done: out ("}"); goto reswitch;
  end
```

This code is used in section 106.

108. The tricky thing about comments is that they might contain I . . . I. We scan ahead for this, and replace the second '*I* ' by a *carriage-return*.

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

```
  begin if cur-type = comment then out2("\")("9");
  id-first ← loc;
  while (loc < limit) A (buffer [loc] ≠ " I ") do incr (loc);
  copy ( id-first );
  if loc < limit then
    begin start_of_line + true; incr (loc); k ← loc;
    while (k < limit) A (buffer[k] ≠ " | ") do incr(k);
    buffer[k] + carriage-return;
    end
  else begin if out-buf [out-ptr] = "\" then out ( "␣");
    out4 ("\")("p")("a")("r"); got0 restart:
    end;
  end
```

This code is used in section 97.

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

```
  begin id-first ← loc; loc + limit; copy(id_first);
  if out-ptr = 0 then
    begin out-ptr + 1; out_buf[1] ← "␣";
    end;
  got0 restart:
  end
```

This code is used in section 97.

110. ⟨ Wind up a line of translation and **goto** *restart*, or finish a |. . . | segment and **goto** *reswitch* 110⟩ ≡
```
begin out("$");
if (loc < limit) ∧ (cur_type = end_of_line) then
  begin cur_type ← recomment; goto reswitch;
  end
else begin out4("\")("p")("a")("r"); goto restart;
  end;
end
```
This code is used in section 97.

111. ⟨Change the translation format of tokens, and **goto** *restart* or *reswitch* 111⟩ ≡
```
begin start_of_line ← false; get_next; t ← cur_type;
while cur_type ≥ min_symbolic_token do
  begin get_next;
  if cur_type ≥ min_symbolic_token then ilk[cur_tok] ← t;
  end;
if cur_type ≠ end_of_line then
  if cur_type ≠ mft_comment then
    begin err_print('!␣Only␣symbolic␣tokens␣should␣appear␣after␣%%%'); goto reswitch
    end;
empty_buffer ← true; goto restart;
end
```
This code is used in section 97.

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

```
  begin initialize;  { beginning of the main program }
  print_ln(banner);  { print a "banner line" }
  ⟨Store all the primitives 65⟩;
  ⟨Store all the translations 73⟩;
  ⟨Initialize the input system 44⟩;
  do_the_translation; ⟨Check that all changes have been read 49⟩;
end_of_MFT:  { here files should be closed if the operating system requires it }
  ⟨Print the job history 113⟩;
  end.
```

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

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

This code is used in section 112.

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

115. **Index.**

```
%%%% MFT commands for the PLAIN base
%%% }( ) 13  ! ↑ %%%% tokens that need no special formatting
%%% step upto donnto %%%% boldface binary operators
%%% addto fill unfill draw undraw %%%% boldface unary operators
%%% addto filldraw unfilldraw drawdot undrawdot erase pickup
%%% addto exitunless stop incr decr proofrulethickness screenrule
%%% addto define-pixels define-whole-pixels define-whole-vertical-pixels
%%%addto define-blacker-pixels define_whole_blacker_pixels
%%% addto define-corrected-pixels lowres_fix proofoffset penstroke
%%% addto beginchar italcorr font-size font-slant labels
%%% addto font-normal-space font-normal-stretch font-normal-shrink font-quad
%%% addto font-x-height font-extra-space font-identifier font-coding-scheme
%%% enddef endchar %%%% boldface closing
%%% true relax mode-setup %%%% boldface nullary operators
%%% true clearit shipit cullit openit showit clearxy clearpen
%%% true nodisplay notransforms screenchars screenstrokes imagerules
%%% . . . . . -- --- %%%% path operators made of dots and dashes
%%% length flex abs dir %%%% unary operators to be in roman type
%%% length unitvector inverse ceiling round vround counterclockwise
%%% length tensepath byte reflectedabout rotatedaround magstep max min
%%% and mod dotprod intersectionpoint softjoin %%%% binary operators to be roman
%%% ++ ** %%%% binary operators made of two special characters
%%% penoffset goodval direction directionpoint %%%% operators that take "of"
%%% pausing tolerance pixels-per-inch blacker o-correction %%%% internals
%%% pausing screen-rows screen,cols currentwindow displaying
%%% pausing pen-top pen,bot pen_lft pen,rt rt lft top bot
%%% = ≤ ≥ ≠ %%%% conversions for the SAIL character set only
```

```
%%%% MFT commands for the Computer Modern base
%%% } ( ) ]] {{ }} ! t %%%% tokens that need no special formatting
%%% step upto downto %%%% boldface binary operators
%%% addto fill unfill draw undraw %%%% boldface unary operators
%%% addto filldraw unfilldraw drawdot undrawdot erase pickup
%%% addto exitunless stop incr decr proofrulethickness screenrule
%%% addto define-pixels define-whole-pixels define_whole_vertical_pixels
%%% addto define-blacker-pixels define-whole-blacker-pixels
%%% addto define-corrected-pixels lowres_fix proofoffset penstroke
%%% addto beginchar beginarithchar italcorr font-size font-slant
%%% addto font-normal-space font-normal-stretch font-normal-shrink font-quad
%%% addto font-x-height font-extra-space font-identifier font-coding-scheme
%%% addto cmchar iff generate adjust-fit math-fit labels penlabels
%%% addto stroke circ,stroke padded
%%% enddef endchar %%%% boldface closing
%%% true relax mode-setup font-setup %%%% boldface nullary operators
%%% true clearit shipit cullit openit showit clearxy clearpen
%%% true nodisplay notransforms screenchars screenstrokes imagerules
%%% ..........-- --- %%%% path operators made of dots and dashes
%%% length flex abs dir %%%% unary operators to be in roman type
%%% length unitvector inverse ceiling hround vround Vround counterclockwise
%%% length tensepath byte reflectedabout rotatedaround magstep max min
%%% and mod dotprod intersectionpoint softjoin %%%% binary operators to be roman
%%% ++ ** %%%% binary operators made of two special characters
%%% penoffset goodval direction directionpoint %%%% operators that take "of"
%%% pausing tolerance pixels-per-inch blacker o-correction %%%% internals
%%% pausing screen-rows screen,cols currentwindow displaying
%%% pausing pen-top pen,bot pen-lit pen,rt shrink-fit rt lft top bot
%%% = ≤ ≥ ≠ %%%% conversions for the SAIL character set only
%%% good crisp fine tiny rule light-rule cal light_cal med,cal heavy_cal
%%% good term fudged mfudged sloped-serif tilted med,tilted
%%% pausing slant fudge math-spread superness superpull beak-darkness ligs
%%% input generate
%% \outer\def↑↑L{\par\vfill\eject} % obeypages
%%
%% % nine-point type:
%% \catcode'@=11 % borrow the private macros of PLAIN (with care)
%% \def\ninebig#1{{\hbox{$\textfont0=\tenrm\textfont2=\tensy
%%  \left#1\vbox  to7.25pt{}\right.\n@space$}}}
%% \catcode'@=12 % now 0 is a nonletter again
%% \font\ninerm=cmr9 \font\sixrm=cmr6
%% \font\ninei=cmmi9 \font\sixi=cmmi6
%% \skewchar\ninei='177 \skewchar\sixi='177
%% \font\ninesy=cmsy9 \font\sixsy=cmsy6
%% \skewchar\ninesy='60 \skewchar\sixsy='60
%% \font\nineit=cmti9
%% \font\ninesl=cmsl9
%% \font\ninebf=cmbx9 \font\sixbf=cmbx6
%% \font\ninett=cmtt9 \hyphenchar\ninett=-1
%% \font\ninetex=cmtex9 \hyphenchar\ninetex=-1
%% \def\rm{\fam0\ninerm}
%% \textfont0=\ninerm \scriptfont0=\sixrm \scriptscriptfont0=\fiverm
%%\textfont1=\ninei \scriptfont1=\sixi \scriptscriptfont1=\fivei
%% \textfont2=\ninesy \scriptfont2=\sixsy \scriptscriptfont2=\fivesy
%% \textfont3=\tenex \scriptfont3=\tenex \scriptscriptfont3=\tenex
%% \def\it{\fam\itfam\nineit}
%% \textfont\itfam=\nineit
%% \def\sl{\fam\slfam\ninesl}
%% \textfont\slfam=\ninesl
%% \def\bf{\fam\bffam
%% \def\_{\kern.04em\vbox{\hrule width.3em height .6pt}\kern.08em}%
%%  \ninebf}
%% \textfont\bffam=\ninebf \scriptfont\bffam=\sixbf
%%  \scriptscriptfont\bffam=\fivebf
%% \def\tt{\fam\ttfam\ninett}
%% \textfont\ttfam=\ninett
%% \def\finstring"#1"{\ninetex"#1"\egroup}
%% \baselineskip=11pt
%% \def\MF{{\manual hijk}\-{\manual lmnj}}
%% \let\big=\ninebig
%% \setbox\strutbox=\hbox{\vrule height8pt depth3pt width0pt}
%% \rm
%% \setbox\shorthyf=\hbox{-\kern-.05em}
%% \hsize=29pc % this is the size of pages in the Computer Modern book
%% \vsize=44pc % likewise
%%
%%%% \mag=\magstep1 %%%% for magnified proofs
```

```
% special macros for use with MFT output

\font\tenlogo=logo10 % font used for the METAFONT logo
\font\tentex=cmtex10 \hyphenchar\tentex=-1 % font used for strings
\font\sevenit=cmti7 \scriptfont\itfam=\sevenit
\def\MF{{\tenlogo META}\-{\tenlogo FONT})}

\parindent=0pt
\thinmuskip=Smu
\thickmuskip=6mu plus 6mu

\def\\#1{{\it#1}} % italic type for identifiers
\def\0#1#2#3{\hbox{\rm\'{}\kern-.2em\it#1#2#3\/\kern.05em}} % octal constant
\def\1#1{\mathop{\hbox{\rm#1}}} % operator, in roman type
\def\2#1{\mathop{\hbox{\bf#1\/\kern.05em}}} % operator, in bold type
\def\3#1{\,\mathclose{\hbox{\bf#1\/}}} % ‘fi’ and ‘endgroup’
\def\4#1{\mathbin{\hbox{\bf#1\/}}} % 'step' and 'at'
\def\5#1{\hbox{\bf#1\/}} % 'true' and 'nullpicture'
\def\6#1{\mathbin{\rm#1}} % ‘++’ and 'scaled'
\def\7{\hbox\bgroup\nocats\frenchspacing\finstring} % string token
\def\8#1{\mathrel{\mathcode‘\.="8000 \mathcode‘\-="8000
  #1\unkern}} % ‘..’ and ‘--’
\def\9{\hfill$\%} % comment separator
\def\?{\mathopen:\;} % colon
\def\frac#1/#2{\leavevmode\kern.1em
  \raise.5ex\hbox{\the\scriptfont0 #1}\kern-.1em
  /\kern-.15em\lower.25ex\hbox{\the\scriptfont0 #2}}

\mathchardef\AM="2026 % ampersand
\let\BL=\medskip % space for empty line
\mathchardef\BS="026E % backslash
\mathchardef\HA="0222 % hat ("005E not as good)
\def\PS{\mathbin{+{-}+}} % Pythagorean subtraction
\def\SH{\raise.7ex\hbox{$\scriptstyle\#$}} % sharp sign for sharped units
\mathchardef\TI="007E % tilde

\chardef\other=12
\def\nocats{\catcode‘\\=\other \catcode‘\{=\other
  \catcode‘\}=\other \catcode‘\$=\other \catcode‘\&=\other
  \catcode‘\#=\other \catcode‘\%=\other \catcode‘\~=\other
  \catcode‘\_=\other \catcode‘\↑=\other}
\def\finstring"#1"{\tentex"#1"\egroup}

\newbox\shorthyf \setbox\shorthyf=\hbox{-\kern-.05em}
\mathchardef\period=‘\.
{\catcode‘\-=\active \global\def-{\copy\shorthyf\mkern3.9mu}
 \catcode‘\.=\active \global\def.{\period\mkern3mu}}

\def\bf{\fam\bffam
  \def\_{\kern.04em\vbox{\hrule width.3em height .6pt}\kern.08em}%
  \tenbf}

\def\join#1${} % say %%\join in .mf file to join lines together
\def\]{\hskip0pt plus 1filll\ } % say % comment\] to get comment flush left
```