The GFtype processor

(Version 3.1, March 1991)

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␣3.1´ { 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

$$(max\_row + 1) \times (max\_col + 1).$$

(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* or greater than *´176*. 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*(*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 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 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. 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 number 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_1 through *paint_63* (opcodes 1 to 63). These are *paint* commands with $d = 1$ to 63, defined as follows: If *paint_switch* = *black*, blacken d pixels of the current row n, in columns m through $m + d - 1$ inclusive. Then, in any case, complement the *paint_switch* and advance m by d.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

post 248. Beginning of the postamble, see below.

post_post 249. Ending of the postamble, see below.

Commands 250–255 are undefined at the present time.

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

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 *skip1* = 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 postamble, which summarizes important facts that METAFONT has accumulated. The postamble has the form

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

Here p is a pointer to the byte following the final *eoc* in the file (or to the byte following the preamble, if there are no characters); it can be used to locate the beginning of *xxx* commands that might have preceded the postamble. The *ds* and *cs* parameters give the design size and check sum, respectively, 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 ≤ -50. If there are no characters in the file, it's possible to have *min_m* $>$ *max_m* and/or *min_n* $>$ *max_n*.)

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

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

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

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

Pointer p might be -1 if the character exists in the TFM file but not in the GF file. This unusual situation can arise in METAFONT output if the user had *proofing* < 0 when the character was being shipped out, but then made *proofing* ≥ 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). METAFONT puts out four to seven of these trailing bytes, until the total length of the file is a multiple of four bytes, since this works out best on machines that pack four bytes per word; but any number of 223's is allowed, as long as there are at least four of them. In effect, 223 is a sort of signature that is added at the very end.

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

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

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

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_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, 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_ln* 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 input_ln;  { inputs a line from the terminal }
  var k: 0 .. terminal_line_length;
  begin update_terminal; reset(term_in);
  if eoln(term_in) then read_ln(term_in);
  k ← 0;
  while (k < terminal_line_length) ∧ ¬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") ∧ (c ≤ "Z") then lower_casify ← c + "a" − "A"
  else lower_casify ← 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_ln(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_ln(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_ln;
  buffer[0] ← lower_casify(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_ln('.')
```

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;
  max_subrow ← 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_0): first_par ← o − paint_0;
  paint1, skip1, char_loc, char_loc + 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_0), sixty_four_cases(new_row_0 + 64), thirty_seven_cases(new_row_0 + 128):
        first_par ← o − new_row_0;
  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␣file␣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 o 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(´undefined␣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_nl; 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 o > 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 ≤ max_subrow then
      begin l ← m − p; r ← m − 1;
      if r > max_subcol then r ← max_subcol;
      m ← l;
      while m ≤ r do
        begin image ← black; incr(m);
        end;
      m ← 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_loc ← cur_loc − 1; print('Postamble starts at byte ', post_loc : 1);
  if post_loc = 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_scaled(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_ln('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_ln('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_loc: 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;
  max_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!');
  for k ← 0 to 255 do
    if char_ptr[k] > 0 then error('missing␣locator␣for␣character␣', k : 1, '!');
  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) ∧ ¬eof(gf_file) do m ← get_byte;
  if ¬eof(gf_file) then bad_gf('signature␣in␣byte␣', cur_loc − 1 : 1, '␣should␣be␣223')
  else if cur_loc < k + 4 then error('not␣enough␣signature␣bytes␣at␣end␣of␣file!')
```

This code is used in section 61.

65.

```
⟨ Process the character locations in the postamble 65 ⟩ ≡
  repeat a ← cur_loc; k ← get_byte;
    if (k = char_loc) ∨ (k = char_loc + 1) then
      begin c ← first_par(k);
      if k = char_loc then
        begin u ← signed_quad; v ← signed_quad;
        end
      else begin u ← get_byte ∗ unity; v ← 0;
        end;
      w ← signed_quad; p ← signed_quad; print('Character␣', c : 1, ':␣dx␣', u : 1, '␣(');
      print_scaled(u);
      if v ≠ 0 then
        begin print('),␣dy␣', v : 1, '␣('); print_scaled(v);
        end;
      print('),␣width␣', w : 1, '␣('); w ← round(w ∗ pix_ratio); print_scaled(w);
      print_ln('),␣loc␣', p : 1);
      if char_ptr[c] = 0 then error('duplicate␣locator␣for␣this␣character!')
      else if p ≠ char_ptr[c] then error('character␣location␣should␣be␣', char_ptr[c] : 1, '!');
      char_ptr[c] ← 0; k ← no_op;
      end;
  until k ≠ no_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(´identification␣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_loc; ⟨ 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_loc − 1 : 1, ´␣is␣not␣boc␣(´, o : 1, ´)´);
      print_nl; print(cur_loc − 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 o = 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.