The GFtoDVI processor

(Version 3.0, October 1989)

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

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

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

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

```
define my_name ≡ 'gftodvi'
define banner ≡ 'This␣is␣GFtoDVI,␣Version␣3.0'  { printed when the program starts }
```

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(stdout,#)
define print_ln(#) ≡ write_ln(stdout,#)
define print_nl(#) ≡ begin write_ln(stdout); write(stdout,#); end
program GF_to_DVI(output);
  const ⟨ Constants in the outer block 5 ⟩
  type ⟨ Types in the outer block 9 ⟩
  var ⟨ Globals in the outer block 12 ⟩
    ⟨ Define parse_arguments 222* ⟩
  procedure initialize; { this procedure gets things started properly }
    var i,j,m,n: integer; { loop indices for initializations }
    begin kpse_set_program_name(argv[0],my_name); kpse_init_prog('GFTODVI',0,nil,nil);
    parse_arguments;
    if verbose then
      begin print(banner); print_ln(version_string);
      end;
    ⟨ Set initial values 13 ⟩
    end;
```

4.* This module deleted, since it only defined the label *final_end*.

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.

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

11.* The original Pascal compiler was designed in the late 60s, when six-bit character sets were common, so it did not make provision for lowercase letters. Nowadays, of course, we need to deal with both capital and small letters in a convenient way. So we shall assume that the Pascal system being used for `GFtoDVI` has a character set containing at least the standard visible ASCII characters (`"!"` through `"~"`). If additional characters are present, `GFtoDVI` can be configured to work with them too.

Some Pascal compilers use the original name *char* for the data type associated with the characters in text files, while other Pascals consider *char* to be a 64-element subrange of a larger data type that has some other name. In order to accommodate this difference, we shall use the name *text_char* to stand for the data type of the characters in the output file. We shall also assume that *text_char* consists of the elements *chr*(*first_text_char*) through *chr*(*last_text_char*), inclusive. The following definitions should be adjusted if necessary.

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

```
⟨ Types in the outer block 9 ⟩ +≡
  text_file = packed file of text_char;
```

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

```
for i ← 0 to '37 do xchr[i] ← chr(i);
```

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

```
⟨ Set initial values 13 ⟩ +≡
  for i ← 1 to '37 do xchr[i] ← chr(i);
  for i ← '177 to '377 do xchr[i] ← chr(i);
```

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

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

```
define update_terminal ≡ fflush(stdout) { empty the terminal output buffer }
define term_in ≡ stdin { standard input }
```

```
⟨ Globals in the outer block 12 ⟩ +≡
buffer: array [0 .. terminal_line_length] of 0 .. 255;
```

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

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

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.

In C, we do path searching based on the user's environment or the default path. We also read the command line and print the banner here (since we don't want to print the banner if the command line is unreasonable).

```
procedure open_gf_file; { prepares to read packed bytes in gf_file }
  begin gf_file ← kpse_open_file(stringcast(name_of_file), kpse_gf_format); cur_loc ← 0;
  end;
procedure open_tfm_file; { prepares to read packed bytes in tfm_file }
  begin tfm_file ← kpse_open_file(stringcast(name_of_file), kpse_tfm_format);
  end;
procedure open_dvi_file; { prepares to write packed bytes in dvi_file }
  begin rewritebin(dvi_file, stringcast(name_of_file));
  end;
```

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

```
⟨ Globals in the outer block 12 ⟩ +≡
cur_loc: integer; { current byte number in gf_file }
name_of_file: ↑text_char;
```

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

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

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

⟨ Types in the outer block 9 ⟩ +≡
font_index $= 0 \,..\, $ *font_mem_size*; *quarterword* $=$ *min_quarterword* $\,..\,$ *max_quarterword*; { 1/4 of a word }
four_quarters $=$ **packed record** *B0*: *quarterword*;
B1: *quarterword*;
B2: *quarterword*;
B3: *quarterword*;
end;
`@\`
`#include␣"gftodmem.h";` `@\` { note the ; so *web2c* will translate types that come after this }
internal_font_number $=$ *title_font* $\,..\,$ *logo_font*;

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

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

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

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

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

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

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

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

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

define $char\_info\_end(\#) \equiv \#\ ].qqqq$
define $char\_info(\#) \equiv font\_info\ [\ char\_base[\#] + char\_info\_end$
define $char\_width\_end(\#) \equiv \#.B0\ ].sc$
define $char\_width(\#) \equiv font\_info\ [\ width\_base[\#] + char\_width\_end$
define $char\_exists(\#) \equiv (\#.B0 > min\_quarterword)$
define $char\_italic\_end(\#) \equiv (qo(\#.B2))$ **div** $4\ ].sc$
define $char\_italic(\#) \equiv font\_info\ [\ italic\_base[\#] + char\_italic\_end$
define $height\_depth(\#) \equiv qo(\#.B1)$
define $char\_height\_end(\#) \equiv (\#)$ **div** $16\ ].sc$
define $char\_height(\#) \equiv font\_info\ [\ height\_base[\#] + char\_height\_end$
define $char\_depth\_end(\#) \equiv \#$ **mod** $16\ ].sc$
define $char\_depth(\#) \equiv font\_info\ [\ depth\_base[\#] + char\_depth\_end$
define $char\_tag(\#) \equiv ((qo(\#.B2))$ **mod** $4)$
define $skip\_byte(\#) \equiv qo(\#.B0)$
define $next\_char(\#) \equiv \#.B1$
define $op\_byte(\#) \equiv qo(\#.B2)$
define $rem\_byte(\#) \equiv \#.B3$

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

```
define store_four_quarters(#) ≡
    begin read_tfm_word; qw.B0 ← qi(b0); qw.B1 ← qi(b1); qw.B2 ← qi(b2);
    qw.B3 ← qi(b3); # ← qw;
    end
```

⟨ Read the `TFM` header 62* ⟩ ≡

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

This code is used in section 59.

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

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

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

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;
  othercases abort('internal␣error');
  endcases;
  ⟨ 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;
```

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;
  othercases abort('internal␣error');
  endcases;
  for j ← 1 to k do cur_gf ← get_byte;
done: cur_gf ← get_byte;
  end;
```

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

```
⟨ Initialize the strings 77 ⟩ +≡
  l ← 0; init_str0(home_font_area);
```

90* And here's the second.

```
function more_name(c : ASCII_code): boolean;
  begin if c = "␣" then more_name ← false
  else begin if c = "/" then
      begin area_delimiter ← pool_ptr; ext_delimiter ← 0;
      end
    else if c = "." then ext_delimiter ← pool_ptr;
    str_room(1); append_char(c); { contribute c to the current string }
    more_name ← true;
    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); 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: integer;
  begin name_length ← length(a) + length(n) + length(e);
  name_of_file ← xmalloc_array(ASCII_code, name_length); k ← −1; { C strings start at position zero. }
  for j ← str_start[a] to str_start[a + 1] − 1 do append_to_name(str_pool[j]);
  for j ← str_start[n] to str_start[n + 1] − 1 do append_to_name(str_pool[j]);
  for j ← str_start[e] to str_start[e + 1] − 1 do append_to_name(str_pool[j]);
  name_of_file[name_length] ← 0;
  end;
```

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

```
procedure start_gf;
  label done;
  var arg_buffer: c_string; arg_buf_ptr: integer;
  begin arg_buffer ← cmdline(optind); arg_buf_ptr ← 0;
  while (line_length < terminal_line_length) ∧ (arg_buffer[arg_buf_ptr] ≠ 0) do
    begin buffer[line_length] ← xord[ucharcast(arg_buffer[arg_buf_ptr])]; incr(line_length);
    incr(arg_buf_ptr);
    end;
  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;
    end;
  job_name ← cur_name; pack_file_name(job_name, null_string, dvi_ext); open_dvi_file;
  end;
```

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

In C, we use a macro to call *fwrite* or *write* directly, writing all the bytes in one shot. Much better even than writing four bytes at a time.

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_ptr > ("7FFFFFFF − dvi_offset) then abort('dvi␣length␣exceeds␣"7FFFFFFF');
  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 > ("7FFFFFFF − dvi_offset) then abort('dvi␣length␣exceeds␣"7FFFFFFF');
  if dvi_ptr > 0 then write_dvi(0, dvi_ptr − 1)
```

This code is used in section 115*.

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 ⟩
```

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* ⟩;
  if verbose then print_ln(´␣´);
  uexit(0);
  end
```

This code is used in section 219*.

118*. ⟨ Set initial values 13 ⟩ +≡
dummy_info.B0 ← *qi*(0); *dummy_info.B1* ← *qi*(0); *dummy_info.B2* ← *qi*(0); *dummy_info.B3* ← *qi*(0);

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

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

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

```
⟨ Process a character 164* ⟩ ≡
  begin check_fonts; ⟨ Finish reading the parameters of the boc 165 ⟩;
  ⟨ Get ready to convert METAFONT coordinates to DVI coordinates 170* ⟩;
  ⟨ Output the bop and the title line 172 ⟩;
  if verbose then
    begin print('[', total_pages : 1); update_terminal;   { print a progress report }
    end;
  ⟨ Output all rules for the current character 173 ⟩;
  ⟨ Output all labels for the current character 181 ⟩;
  do_pixels; dvi_out(eop);   { finish the page }
  ⟨ Adjust the maximum page width 203 ⟩;
  if verbose then
    begin print(']');
    if total_pages mod 13 = 0 then print_ln('␣')
    else print('␣');
    update_terminal;
    end;
  end
```

This code is used in section 219*.

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

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

This code is used in section 164*.

215* **define** *do_skip* ≡ *z* ← 0; *paint_black* ← *false*
define *end_with*(#) ≡
begin #; *cur_gf* ← *get_byte*; **goto** *done1*; **end**
define *five_cases*(#) ≡ #, # + 1, # + 2, # + 3, # + 4
define *eight_cases*(#) ≡ #, # + 1, # + 2, # + 3, # + 4, # + 5, # + 6, # + 7
define *thirty_two_cases*(#) ≡ *eight_cases*(#), *eight_cases*(# + 8), *eight_cases*(# + 16), *eight_cases*(# + 24)
define *sixty_four_cases*(#) ≡ *thirty_two_cases*(#), *thirty_two_cases*(# + 32)

⟨ Read and process GF commands until coming to the end of this row 215* ⟩ ≡
loop begin *continue*: **if** (*cur_gf* ≥ *new_row_0*) ∧ (*cur_gf* ≤ *new_row_0* + 164) **then**
end_with(*z* ← *cur_gf* − *new_row_0*; *paint_black* ← *true*)
else case *cur_gf* **of**
sixty_four_cases(0): *k* ← *cur_gf*;
paint1: *k* ← *get_byte*;
paint2: *k* ← *get_two_bytes*;
paint3: *k* ← *get_three_bytes*;
eoc: **goto** *done1*;
skip0: *end_with*(*blank_rows* ← 0; *do_skip*);
skip1: *end_with*(*blank_rows* ← *get_byte*; *do_skip*);
skip2: *end_with*(*blank_rows* ← *get_two_bytes*; *do_skip*);
skip3: *end_with*(*blank_rows* ← *get_three_bytes*; *do_skip*);
xxx1, *xxx2*, *xxx3*, *xxx4*, *yyy*, *no_op*: **begin** *skip_nop*; **goto** *continue*;
end;
othercases *bad_gf*(´Improper␣opcode´)
endcases;
⟨ Paint *k* bits and read another command 216 ⟩;
end;
done1:

This code is used in section 214.

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

```
begin initialize; { get all variables initialized }
⟨ Initialize the strings 77 ⟩;
start_gf; { open the input and output files }
⟨ Process the preamble 221 ⟩;
cur_gf ← get_byte; init_str_ptr ← str_ptr;
loop begin ⟨ Initialize variables for the next character 144 ⟩;
  while (cur_gf ≥ xxx1) ∧ (cur_gf ≤ no_op) do ⟨ Process a no-op command 154 ⟩;
  if cur_gf = post then ⟨ Finish the DVI file and goto final_end 115* ⟩;
  if cur_gf ≠ boc then
    if cur_gf ≠ boc1 then abort('Missing␣boc!');
  ⟨ Process a character 164* ⟩;
  cur_gf ← get_byte; str_ptr ← init_str_ptr; pool_ptr ← str_start[str_ptr];
  end;
if verbose ∧ (total_pages mod 13 ≠ 0) then print_ln('␣');
end.
```

222* System-dependent changes. Parse a Unix-style command line.

```
define argument_is(#) ≡ (strcmp(long_options[option_index].name, #) = 0)
⟨ Define parse_arguments 222* ⟩ ≡
procedure parse_arguments;
  const n_options = 4;  { Pascal won't count array lengths for us. }
  var long_options: array [0 .. n_options] of getopt_struct;
    getopt_return_val: integer; option_index: c_int_type; current_option: 0 .. n_options;
  begin ⟨ Initialize the option variables 227* ⟩;
  ⟨ Define the option table 223* ⟩;
  repeat getopt_return_val ← getopt_long_only(argc, argv, ´´, long_options, address_of(option_index));
    if getopt_return_val = −1 then
      begin do_nothing;  { End of arguments; we exit the loop below. }
      end
    else if getopt_return_val = "?" then
        begin usage(my_name);
        end
      else if argument_is(´help´) then
          begin usage_help(GFTODVI_HELP, nil);
          end
        else if argument_is(´version´) then
            begin print_version_and_exit(banner, nil, ´D.E.␣Knuth´, nil);
            end
          else if argument_is(´overflow-label-offset´) then
              begin offset_in_points ← atof(optarg);
              overflow_label_offset ← round(offset_in_points * 65536);
              end;  { Else it was a flag; getopt has already done the assignment. }
  until getopt_return_val = −1;  { Now optind is the index of first non-option on the command line. We
        must have one remaining argument. }
  if (optind + 1 ≠ argc) then
    begin write_ln(stderr, my_name, ´:␣Need␣exactly␣one␣file␣argument.´); usage(my_name);
    end;
  end;
```

This code is used in section 3*.

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

```
⟨ Define the option table 223* ⟩ ≡
  current_option ← 0; long_options[current_option].name ← ´help´;
  long_options[current_option].has_arg ← 0; long_options[current_option].flag ← 0;
  long_options[current_option].val ← 0; incr(current_option);
```

See also sections 224*, 225*, 228*, and 231*.

This code is used in section 222*.

224* Another of the standard options.

```
⟨ Define the option table 223* ⟩ +≡
  long_options[current_option].name ← ´version´; long_options[current_option].has_arg ← 0;
  long_options[current_option].flag ← 0; long_options[current_option].val ← 0; incr(current_option);
```

225* Print progress information?

⟨ Define the option table 223* ⟩ +≡
long_options[*current_option*].*name* ← ´verbose´; *long_options*[*current_option*].*has_arg* ← 0;
long_options[*current_option*].*flag* ← *address_of*(*verbose*); *long_options*[*current_option*].*val* ← 1;
incr(*current_option*);

226* ⟨ Globals in the outer block 12 ⟩ +≡
verbose: *c_int_type*;

227* ⟨ Initialize the option variables 227* ⟩ ≡
verbose ← *false*;

See also section 230*.

This code is used in section 222*.

228* Change how far from the right edge of the character boxes we print overflow labels.

⟨ Define the option table 223* ⟩ +≡
long_options[*current_option*].*name* ← ´overflow-label-offset´;
long_options[*current_option*].*has_arg* ← 1; *long_options*[*current_option*].*flag* ← 0;
long_options[*current_option*].*val* ← 0; *incr*(*current_option*);

229* It's easier on the user to specify the value in TEX points, but we want to store it in scaled points.

⟨ Globals in the outer block 12 ⟩ +≡
overflow_label_offset: *integer*; { in scaled points }
offset_in_points: *real*;

230* The default offset is ten million scaled points—a little more than two inches.

⟨ Initialize the option variables 227* ⟩ +≡
overflow_label_offset ← 10000000;

231* An element with all zeros always ends the list.

⟨ Define the option table 223* ⟩ +≡
long_options[*current_option*].*name* ← 0; *long_options*[*current_option*].*has_arg* ← 0;
long_options[*current_option*].*flag* ← 0; *long_options*[*current_option*].*val* ← 0;

232* 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 following sections were changed by the change file: 1, 3, 4, 8, 11, 14, 16, 17, 47, 48, 52, 55, 62, 78, 81, 85, 88, 90, 92, 94, 107, 108, 109, 111, 115, 118, 138, 164, 170, 215, 219, 222, 223, 224, 225, 226, 227, 228, 229, 230, 231, 232.

⟨ Process the preamble 221 ⟩ Used in section 219*.
⟨ Put the bits for the next row, times l, into a 214 ⟩ Used in section 213.
⟨ Read and check the font data; *abend* if the TFM file is malformed; otherwise **goto** *done* 59 ⟩ Used in section 58.
⟨ Read and process GF commands until coming to the end of this row 215* ⟩ Used in section 214.
⟨ Read box dimensions 64 ⟩ Used in section 59.
⟨ Read character data 63 ⟩ Used in section 59.
⟨ Read extensible character recipes 67 ⟩ Used in section 59.
⟨ Read font parameters 68 ⟩ Used in section 59.
⟨ Read ligature/kern program 66 ⟩ Used in section 59.
⟨ Read the next k characters of the GF file; change c and **goto** *done* if a keyword is recognized 82 ⟩ Used in section 81*.
⟨ Read the TFM header 62* ⟩ Used in section 59.
⟨ Read the TFM size fields 60 ⟩ Used in section 59.
⟨ Remove all rectangles from list, except for dots that have labels 201 ⟩ Used in section 200.
⟨ Replace z by z' and compute α, β 65 ⟩ Used in section 64.
⟨ Scan the file name in the buffer 95 ⟩ Used in section 94*.
⟨ Search buffer for valid keyword; if successful, **goto** *found* 100 ⟩ Used in section 99.
⟨ Search for the nearest dot in nodes following p 151 ⟩ Used in section 150.
⟨ Search for the nearest dot in nodes preceding p 152 ⟩ Used in section 150.
⟨ Set initial values 13, 14*, 15, 54, 97, 103, 106, 118*, 126, 142 ⟩ Used in section 3*.
⟨ Store a label 163 ⟩ Used in section 154.
⟨ Store a rule 159 ⟩ Used in section 154.
⟨ Store a title 162 ⟩ Used in section 154.
⟨ Try the first choice for label direction 196 ⟩ Used in section 195.
⟨ Try the fourth choice for label direction 199 ⟩ Used in section 195.
⟨ Try the second choice for label direction 197 ⟩ Used in section 195.
⟨ Try the third choice for label direction 198 ⟩ Used in section 195.
⟨ Try to output a diagonal rule 178 ⟩ Used in section 173.
⟨ Types in the outer block 9, 10, 11*, 45, 52*, 70, 79, 104, 139 ⟩ Used in section 3*.
⟨ Typeset a space in font f and advance k 119 ⟩ Used in section 116.
⟨ Typeset an overflow label for p 202 ⟩ Used in section 200.
⟨ Typeset character *cur_l*, if it exists in the font; also append an optional kern 121 ⟩ Used in section 116.
⟨ Typeset the pixels of the current row 208 ⟩ Used in section 218.
⟨ Update the font name or area 101 ⟩ Used in section 99.
⟨ Use size fields to allocate font information 61 ⟩ Used in section 59.
⟨ Vertically typeset p copies of character $k+1$ 180 ⟩ Used in section 178.
⟨ Vertically typeset q copies of character k 179 ⟩ Used in section 178.