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.

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 *my_name* ≡ `'gftype'`
define *banner* ≡ `'This␣is␣GFtype,␣Version␣3.1'` { printed when the program starts }

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*(*stdout*, #)
define *print_ln*(#) ≡ *write_ln*(*stdout*, #)
define *print_nl* ≡ *write_ln*(*stdout*)

```
program GF_type(gf_file, output);
  const ⟨ Constants in the outer block 5* ⟩
  type ⟨ Types in the outer block 8 ⟩
  var ⟨ Globals in the outer block 4* ⟩
    ⟨ Define parse_arguments 73* ⟩
  procedure initialize; { this procedure gets things started properly }
    var i: integer; { loop index for initializations }
      bound_default: integer; { temporary for setup }
      bound_name: const_cstring; { temporary for setup }
    begin kpse_set_program_name(argv[0], my_name); kpse_init_prog('GFTYPE', 0, nil, nil);
    parse_arguments; print(banner); print_ln(version_string); ⟨ Set initial values 6* ⟩
    end;
```

4* This module is deleted, because it is only useful for a non-local goto, which we can't use in C.

Instead, we define parameters settable at runtime.

```
⟨ Globals in the outer block 4* ⟩ ≡
line_length: integer; { xxx strings will not produce lines longer than this }
max_rows: integer; { largest possible vertical extent of pixel image array }
max_cols: integer; { largest possible horizontal extent of pixel image array }
max_row: integer; { current vertical extent of pixel image array }
max_col: integer; { current horizontal extent of pixel image array }
```

See also sections 10, 21, 23, 25*, 35, 37*, 41, 46, 54, 62, and 67.

This code is used in section 3*.

5* Three parameters can be changed at run 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.)

```
define def_line_length = 500 { default line_length value }
define max_image = 8191 { largest possible extent of METAFONT's pixel image array }
⟨ Constants in the outer block 5* ⟩ ≡
  inf_line_length = 20; sup_line_length = 1023;
```

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 }
define const_chk(#) ≡
        begin if # < inf@&# then # ← inf@&#
        else if # > sup@&# then # ← sup@&#
        end { setup_bound_var stuff duplicated in tex.ch. }
define setup_bound_var(#) ≡ bound_default ← #; setup_bound_var_end
define setup_bound_var_end(#) ≡ bound_name ← #; setup_bound_var_end_end
define setup_bound_var_end_end(#) ≡ setup_bound_variable(address_of(#), bound_name, bound_default);
⟨ Set initial values 6* ⟩ ≡
    { See comments in tex.ch for why the name has to be duplicated. }
  setup_bound_var(def_line_length)('line_length')(line_length);
      { xxx strings will not produce lines longer than this }
  setup_bound_var(max_image)('max_rows')(max_rows);
      { largest allowed vertical extent of pixel image array }
  setup_bound_var(max_image)('max_cols')(max_cols);
      { largest allowed horizontal extent of pixel image array }
  const_chk(line_length);
  if max_rows > max_image then max_rows ← max_image;
  if max_cols > max_image then max_cols ← max_image;
  image_array ← nil;
```

See also sections 11, 12, 26*, 47, and 63.

This code is used in section 3*.

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 we might want to *abort* the program with an error message.

```
define abort(#) ≡
        begin write_ln(stderr, #); uexit(1);
        end
define bad_gf(#) ≡ abort('Bad␣GF␣file:␣', #, '!')
```

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 char ≡ 0 .. 255
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;
```

22* In C, we do path searching based on the user's environment or the default path.

```
procedure open_gf_file; { prepares to read packed bytes in gf_file }
  begin gf_file ← kpse_open_file(cmdline(optind), kpse_gf_format); cur_loc ← 0;
  ⟨ Print all the selected options 34* ⟩;
  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 4* ⟩ +≡
wants_mnemonics: c_int_type;  { controls mnemonic output }
wants_pixels: c_int_type;  { controls pixel output }
```

26*. ⟨ Set initial values 6* ⟩ +≡

27*. There is no terminal input. The options for running this program are offered through command line options.

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.)

30*. This was so humdrum that we got rid of it. (module 30)

31*. The dialog procedure module is eliminated. (module 31)

32*. So is its first part. (module 32)

33*. So is its second part. (module 33)

34*. After the command-line switches have been processed, 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 22*.

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* + (*max_col* + 1) ∗ *n*]

⟨ Globals in the outer block 4* ⟩ +≡
image_array: ↑*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_col* ← *max_m_stated* − *min_m_stated* − 1;
if *max_col* > *max_cols* **then** *max_col* ← *max_cols*;
max_row ← *max_n_stated* − *min_n_stated*;
if *max_row* > *max_rows* **then** *max_row* ← *max_rows*;
if (*max_row* ≥ 0) ∧ (*max_col* ≥ 0) **then** *image_array* ← *xcalloc_array*(*pixel*, *max_col*, *max_row*);
end

This code is used in section 71.

39* With *image_array* allocated dynamically these are the same.

define *max_subrow* ≡ *max_row* { vertical size of current subarray of interest }
define *max_subcol* ≡ *max_col* { horizontal 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.)´);
if (*max_row* ≥ 0) ∧ (*max_col* ≥ 0) **then**
begin *libc_free*(*image_array*); *image_array* ← **nil**;
end;
end

This code is used in section 69.

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(xchr[ord('0') + (s div unity)]); s ← 10 * (s mod unity); delta ← delta * 10;
    until s ≤ delta;
    end;
  end;
```

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): first_par ← o − new_row_0;
  sixty_four_cases(new_row_0 + 64): first_par ← o − new_row_0;
  thirty_seven_cases(new_row_0 + 128): first_par ← o − new_row_0;
  othercases abort('internal␣error')
  endcases;
  end;
```

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 $o \le$ *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*): ⟨ Translate a *new_row* command 59 ⟩;
sixty_four_cases(*new_row_0* $+ 64$): ⟨ Translate a *new_row* command 59 ⟩;
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.

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 }
⟨ 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_ln('␣altogether.');
end.
```

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

```
define argument_is(#) ≡ (strcmp(long_options[option_index].name, #) = 0)
define do_nothing ≡ { empty statement }
⟨ Define parse_arguments 73* ⟩ ≡
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 ⟨ Define the option table 74* ⟩;
  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(GFTYPE_HELP, nil);
          end
        else if argument_is(´version´) then
            begin print_version_and_exit(banner, nil, ´D.R.␣Fuchs´, nil);
            end; { Else it was a flag. }
  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*.

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

```
⟨ Define the option table 74* ⟩ ≡
  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 75*, 76*, 77*, and 78*.

This code is used in section 73*.

75* Another of the standard options.

```
⟨ Define the option table 74* ⟩ +≡
  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);
```

76* Translate commands?

```
⟨ Define the option table 74* ⟩ +≡
  long_options[current_option].name ← ´mnemonics´; long_options[current_option].has_arg ← 0;
  long_options[current_option].flag ← address_of(wants_mnemonics);
  long_options[current_option].val ← 1; incr(current_option);
```

77:* Show pixels?

⟨ Define the option table 74* ⟩ +≡
long_options[*current_option*].*name* ← ´`images`´; *long_options*[*current_option*].*has_arg* ← 0;
long_options[*current_option*].*flag* ← *address_of*(*wants_pixels*); *long_options*[*current_option*].*val* ← 1;
incr(*current_option*);

78:* An element with all zeros always ends the list.

⟨ Define the option table 74* ⟩ +≡
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;

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

The following sections were changed by the change file: 1, 3, 4, 5, 6, 7, 9, 22, 25, 26, 27, 29, 30, 31, 32, 33, 34, 37, 38, 39, 40, 45, 48, 51, 66, 73, 74, 75, 76, 77, 78, 79.