The PLtoTF processor

(Version 3.6, January 2014)

	Section	Page
Introduction	1	302
Property list description of font metric data	5	303
Basic input routines	17	304
Basic scanning routines	30	305
Scanning property names	36	305
Scanning numeric data	50	305
Storing the property values	67	305
The input phase	81	306
The checking and massaging phase	109	307
The output phase	127	309
The main program	146	310
System-dependent changes	148	311
Index	156	313

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

1*. Introduction. The `PLtoTF` utility program converts property-list ("`PL`") files into equivalent TeX font metric ("`TFM`") files. It also makes a thorough check of the given `PL` file, so that the `TFM` file should be acceptable to TeX.

The first `PLtoTF` program was designed by Leo Guibas in the summer of 1978. Contributions by Frank Liang, Doug Wyatt, and Lyle Ramshaw also had a significant effect on the evolution of the present code.

Extensions for an enhanced ligature mechanism were added by the author in 1989.

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

```
define my_name ≡ ´pltotf´
define banner ≡ ´This␣is␣PLtoTF,␣Version␣3.6´   { printed when the program starts }
```

2*. This program is written entirely in standard Pascal, except that it has to do some slightly system-dependent character code conversion on input. Furthermore, lower case letters are used in error messages; they could be converted to upper case if necessary. The input is read from *pl_file*, and the output is written on *tfm_file*; 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(stderr, #)
define print_ln(#) ≡ write_ln(stderr, #)
define print_real(#) ≡ fprint_real(stderr, #)
program PLtoTF(pl_file, tfm_file, output);
  const ⟨ Constants in the outer block 3* ⟩
  type ⟨ Types in the outer block 17 ⟩
  var ⟨ Globals in the outer block 5 ⟩
    ⟨ Define parse_arguments 148* ⟩
  procedure initialize;   { this procedure gets things started properly }
    var ⟨ Local variables for initialization 19 ⟩
    begin kpse_set_program_name(argv[0], my_name); parse_arguments; ⟨ Set initial values 6* ⟩
    end;
```

3*. The following parameters can be changed at compile time to extend or reduce `PLtoTF`'s capacity.

```
⟨ Constants in the outer block 3* ⟩ ≡
  buf_size = 3000;   { length of lines displayed in error messages }
  max_header_bytes = 1000;   { four times the maximum number of words allowed in the TFM file header
      block, must be 1024 or less }
  max_param_words = 254;   { the maximum number of fontdimen parameters allowed }
  max_lig_steps = 32510;   { maximum length of ligature program, must be at most 32767 − 257 = 32510 }
  max_kerns = 5000;   { the maximum number of distinct kern values }
  hash_size = 32579;
      { preferably a prime number, a bit larger than the number of character pairs in lig/kern steps }
```

This code is used in section 2*.

6* ⟨ Set initial values 6* ⟩ ≡

```
reset(pl_file, pl_name);
if verbose then
   begin print(banner); print_ln(version_string);
   end;
```

See also sections 16*, 20, 22, 24, 26*, 37, 41, 70, 74, and 119.

This code is used in section 2*.

16* On some systems you may have to do something special to write a packed file of bytes. It's no problem in C.

⟨ Set initial values 6* ⟩ +≡

```
rewritebin(tfm_file, tfm_name);
```

18*. One of the things `PLtoTF` has to do is convert characters of strings to ASCII form, since that is the code used for the family name and the coding scheme in a `TFM` file. An array *xord* is used to do the conversion from *char*; the method below should work with little or no change on most Pascal systems.

```
define char ≡ 0 .. 255
define first_ord = 0  { ordinal number of the smallest element of char }
define last_ord = 127  { ordinal number of the largest element of char }
```

⟨ Globals in the outer block 5 ⟩ +≡
```
xord: array [char] of ASCII_code;  { conversion table }
```

25*. Just before each `CHARACTER` property list is evaluated, the character code is printed in octal notation. Up to eight such codes appear on a line; so we have a variable to keep track of how many are currently there.

⟨ Globals in the outer block 5 ⟩ +≡
```
chars_on_line: 0 .. 8;  { the number of characters printed on the current line }
perfect: boolean;  { was the file free of errors? }
```

26*. ⟨ Set initial values 6* ⟩ +≡
```
  chars_on_line ← 0; perfect ← true;  { innocent until proved guilty }
```

27*. The following routine prints an error message and an indication of where the error was detected. The error message should not include any final punctuation, since this procedure supplies its own.

```
  define err_print(#) ≡
          begin if chars_on_line > 0 then print_ln(´␣´);
          print(#); show_error_context;
          end
procedure show_error_context;  { prints the current scanner location }
  var k: 0 .. buf_size;  { an index into buffer }
  begin print_ln(´␣(line␣´, line : 1, ´).´);
  if ¬left_ln then print(´...´);
  for k ← 1 to loc do print(buffer[k]);  { print the characters already scanned }
  print_ln(´␣´);
  if ¬left_ln then print(´␣␣␣´);
  for k ← 1 to loc do print(´␣´);  { space out the second line }
  for k ← loc + 1 to limit do print(buffer[k]);  { print the characters yet unseen }
  if right_ln then print_ln(´␣´) else print_ln(´...´);
  chars_on_line ← 0; perfect ← false;
  end;
```

79* When we are nearly ready to output the `TFM` file, we will set *index*[*p*] $\leftarrow$ *k* if the dimension in *memory*[*p*] is being rounded to the *k*th element of its list.

```
  define index ≡ index_var
⟨ Globals in the outer block 5 ⟩ +≡
index: array [pointer] of byte;
excess: byte;  { number of words to remove, if list is being shortened }
```

103* Finally we come to the part of `PLtoTF`'s input mechanism that is used most, the processing of individual character data.

```
⟨Read character info list 103*⟩ ≡
  begin c ← get_byte;   { read the character code that is being specified }
  if verbose then ⟨Print c in octal notation 108⟩;
  while level = 1 do
    begin while cur_char = "␣" do get_next;
    if cur_char = "(" then ⟨Read a character property 104⟩
    else if cur_char = ")" then skip_to_end_of_item
      else junk_error;
    end;
  if char_wd[c] = 0 then char_wd[c] ← sort_in(width, 0);   { legitimatize c }
  finish_inner_property_list;
  end
```

This code is used in section 146.

115* **define** *round_message*(#) ≡

```
    if delta > 0 then
      begin print(´I␣had␣to␣round␣some␣´, #, ´s␣by␣´);
      print_real((((delta + 1) div 2)/´4000000), 1, 7); print_ln(´␣units.´);
      end
```

⟨Put the width, height, depth, and italic lists into final form 115*⟩ ≡

```
  delta ← shorten(width, 255); set_indices(width, delta); round_message(´width´);
  delta ← shorten(height, 15); set_indices(height, delta); round_message(´height´);
  delta ← shorten(depth, 15); set_indices(depth, delta); round_message(´depth´);
  delta ← shorten(italic, 63); set_indices(italic, delta); round_message(´italic␣correction´);
```

This code is used in section 110.

117* It's not trivial to check for infinite loops generated by repeated insertion of ligature characters. But fortunately there is a nice algorithm for such testing, copied here from the program `TFtoPL` where it is explained further.

```
  define simple = 0   { $f(x, y) = z$ }
  define left_z = 1   { $f(x, y) = f(z, y)$ }
  define right_z = 2   { $f(x, y) = f(x, z)$ }
  define both_z = 3   { $f(x, y) = f(f(x, z), y)$ }
  define pending = 4   { $f(x, y)$ is being evaluated }
  define class ≡ class_var   { Avoid problems with AIX <math.h> }
```

123* (More good stuff from `TFtoPL`.)

```
  ifdef(´notdef´)
  function f(h, x, y : indx): indx;
    begin end;
    { compute f for arguments known to be in hash[h] }
endif(´notdef´)
function eval(x, y : indx): indx;   { compute $f(x, y)$ with hashtable lookup }
  var key: integer;   { value sought in hash table }
  begin key ← 256 ∗ x + y + 1; h ← (1009 ∗ key) mod hash_size;
  while hash[h] > key do
    if h > 0 then decr(h) else h ← hash_size;
  if hash[h] < key then eval ← y   { not in ordered hash table }
  else eval ← f(h, x, y);
  end;
```

124* Pascal's beastly convention for *forward* declarations prevents us from saying **function** $f(h, x, y : indx)$: *indx* here.

```
function f(h, x, y : indx): indx;
  begin case class[h] of
  simple: do_nothing;
  left_z: begin class[h] ← pending; lig_z[h] ← eval(lig_z[h], y); class[h] ← simple;
    end;
  right_z: begin class[h] ← pending; lig_z[h] ← eval(x, lig_z[h]); class[h] ← simple;
    end;
  both_z: begin class[h] ← pending; lig_z[h] ← eval(eval(x, lig_z[h]), y); class[h] ← simple;
    end;
  pending: begin x_lig_cycle ← x; y_lig_cycle ← y; lig_z[h] ← 257; class[h] ← simple;
    end; { the value 257 will break all cycles, since it's not in hash }
  end; { there are no other cases }
  f ← lig_z[h];
  end;
```

127*. The output phase. Now that we know how to get all of the font data correctly stored in `PLtoTF`'s memory, it only remains to write the answers out.

First of all, it is convenient to have an abbreviation for output to the `TFM` file:

```
define out(#) ≡ putbyte(#, tfm_file)
```

130*. It might turn out that no characters exist at all. But `PLtoTF` keeps going and writes the `TFM` anyway. In this case *ec* will be 0 and *bc* will be 1.

```
⟨Compute the twelve subfile sizes 130*⟩ ≡
  lh ← header_ptr div 4;
  not_found ← true; bc ← 0;
  while not_found do
    if (char_wd[bc] > 0) ∨ (bc = 255) then not_found ← false
    else incr(bc);
  not_found ← true; ec ← 255;
  while not_found do
    if (char_wd[ec] > 0) ∨ (ec = 0) then not_found ← false
    else decr(ec);
  if bc > ec then bc ← 1;
  incr(memory[width]); incr(memory[height]); incr(memory[depth]); incr(memory[italic]);
  ⟨Compute the ligature/kern program offset 139⟩;
  lf ← 6 + lh + (ec − bc + 1) + memory[width] + memory[height] + memory[depth] + memory[italic] + nl +
      lk_offset + nk + ne + np;
  if lf < 0 then
    begin print_ln('The␣total␣number␣of␣words␣in␣the␣TFM␣file␣too␣large!'); uexit(1);
    end
```

This code is used in section 128.

136*. When a scaled quantity is output, we may need to divide it by *design_units*. The following subroutine takes care of this, using floating point arithmetic only if *design_units* $\neq 1.0$.

```
procedure out_scaled(x : fix_word); {outputs a scaled fix_word}
  var n: byte; {the first byte after the sign}
    m: 0 .. 65535; {the two least significant bytes}
  begin if fabs(x/design_units) ≥ 16.0 then
    begin print('The␣relative␣dimension␣'); print_real(x/'4000000, 1, 3);
    print_ln('␣is␣too␣large.'); print('␣␣(Must␣be␣less␣than␣16*designsize');
    if design_units ≠ unity then
      begin print('␣='); print_real(design_units/'200000, 1, 3); print('␣designunits');
      end;
    print_ln(')'); x ← 0;
    end;
  if design_units ≠ unity then x ← round((x/design_units) ∗ 1048576.0);
  if x < 0 then
    begin out(255); x ← x + '100000000;
    if x ≤ 0 then x ← 1;
    end
  else begin out(0);
    if x ≥ '100000000 then x ← '77777777;
    end;
  n ← x div '200000; m ← x mod '200000; out(n); out(m div 256); out(m mod 256);
  end;
```

147* Here is where `PLtoTF` begins and ends.

```
begin initialize;
name_enter;
read_input;
if verbose then print_ln(´.´);
corr_and_check;
⟨Do the output 128⟩;
if ¬perfect then uexit(1);
end.
```

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

```
  define argument_is(#) ≡ (strcmp(long_options[option_index].name, #) = 0)
⟨Define parse_arguments 148*⟩ ≡
procedure parse_arguments;
  const n_options = 3;   { 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 153*⟩;
  ⟨Define the option table 149*⟩;
  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(PLTOTF_HELP, nil);
          end
        else if argument_is(´version´) then
            begin print_version_and_exit(banner, nil, ´D.E.␣Knuth´, nil);
            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 or two remaining arguments. }
  if (optind + 1 ≠ argc) ∧ (optind + 2 ≠ argc) then
    begin write_ln(stderr, my_name, ´:␣Need␣one␣or␣two␣file␣arguments.´); usage(my_name);
    end;
  pl_name ← extend_filename(cmdline(optind), ´pl´);
      { If an explicit output filename isn't given, construct it from pl_name. }
  if optind + 2 = argc then
    begin tfm_name ← extend_filename(cmdline(optind + 1), ´tfm´);
    end
  else begin tfm_name ← basename_change_suffix(pl_name, ´.pl´, ´.tfm´);
    end;
  end;
```

This code is used in section 2*.

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

```
⟨Define the option table 149*⟩ ≡
  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 150*, 151*, and 154*.

This code is used in section 148*.

150*. Another of the standard options.

```
⟨Define the option table 149*⟩ +≡
  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);
```

151* Print progress information?

⟨ Define the option table 149* ⟩ +≡

```
   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);
```

152* ⟨ Globals in the outer block 5 ⟩ +≡

verbose: *c_int_type*;

153* ⟨ Initialize the option variables 153* ⟩ ≡

```
   verbose ← false;
```

This code is used in section 148*.

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

⟨ Define the option table 149* ⟩ +≡

```
   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;
```

155* Global filenames.

⟨ Globals in the outer block 5 ⟩ +≡

tfm_name, *pl_name*: *const_c_string*;

156* 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, 2, 3, 6, 16, 18, 25, 26, 27, 79, 103, 115, 117, 123, 124, 127, 130, 136, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156.

-help: 149*.
-version: 150*.
A cycle of NEXTLARGER...: 113.
acc: 51, 52, 53, 54, 55, 56, 62, 64, 66.
address_of: 148*, 151*.
An "R" or "D" ... needed here: 62.
An octal ("O") or hex ("H")...: 59.
argc: 148*.
argument_is: 148*.
argv: 2*, 148*.
ASCII_code: 17, 18*, 30, 36, 38, 51.
At most 256 VARCHAR specs...: 105.
backup: 32, 53, 54, 55, 97.
bad_indent: 29.
banner: 1*, 6*, 148*.
basename_change_suffix: 148*.
bc: 129, 130*, 131, 134, 135, 140.
bchar: 67, 70, 85, 120, 125, 126, 138, 139, 142.
bchar_label: 72, 74, 97, 110, 116, 125, 139.
boolean: 23, 25*, 42, 62, 67, 98, 109, 121, 129, 138.
BOT piece of character...: 112.
both_z: 117*, 121, 122, 124*.
boundary_char_code: 44, 47, 85.
buf_size: 3*, 23, 27*, 28.
buffer: 23, 27*, 28, 29, 31, 32, 33, 52.
byte: 17, 44, 45, 51, 57, 67, 72, 73, 79*, 80, 81, 87, 96, 107, 129, 136*, 138, 146.
b0: 57, 58, 99, 100, 101, 102, 105, 106, 112, 116, 120, 126, 142, 143.
b1: 57, 58, 101, 102, 105, 106, 112, 116, 120, 122, 126, 142, 143.
b2: 57, 58, 101, 102, 105, 106, 112, 116, 120, 122, 126, 139, 142, 143.
b3: 57, 58, 101, 102, 105, 106, 112, 116, 120, 122, 126, 139, 142, 143.
c: 59, 73, 81, 121, 146.
"C" value must be...: 52.
c_int_type: 148*, 152*.
cc: 121, 122, 138, 140, 141.
char: 18*, 23.
char_dp: 72, 74, 104, 135.
char_dp_code: 44, 47, 104.
char_ht: 72, 74, 104, 135.
char_ht_code: 44, 47, 104.
char_ic: 72, 74, 104, 135.
char_ic_code: 44, 47, 104.
char_info: 135.
char_info_code: 44.
char_info_word: 72.
char_remainder: 72, 74, 97, 104, 105, 111, 112, 113, 120, 125, 135, 138, 140, 141.
char_tag: 72, 74, 96, 97, 104, 105, 111, 113, 125, 135, 140.
char_wd: 72, 74, 75, 103*, 104, 110, 111, 126, 130*, 134, 135.
char_wd_code: 44, 47, 93, 104.
character_code: 44, 47, 84, 85.
chars_on_line: 25*, 26*, 27*, 108.
check_existence: 111, 120.
check_existence_and_safety: 111, 112.
check_sum_code: 44, 47, 85.
check_sum_loc: 70, 85, 134.
check_sum_specified: 67, 70, 85, 133.
check_tag: 96, 97, 104, 105.
chr: 20, 28.
class: 117*, 118, 121, 124*, 125.
class_var: 117*.
clear_lig_kern_entry: 116.
cmdline: 148*.
coding_scheme_code: 44, 47, 85.
coding_scheme_loc: 70, 85.
comment_code: 44, 47, 84, 93, 95, 104, 106.
const_c_string: 155*.
corr_and_check: 146, 147*.
cur_bytes: 57, 58.
cur_char: 30, 31, 32, 33, 34, 35, 49, 51, 52, 53, 54, 55, 56, 59, 60, 62, 63, 64, 66, 82, 84, 87, 90, 92, 94, 97, 103*, 105.
cur_code: 44, 49, 84, 85, 93, 95, 101, 104, 106.
cur_hash: 39, 42, 43, 45.
cur_name: 38, 42, 43, 45, 46, 49.
current_option: 148*, 149*, 150*, 151*, 154*.
c0: 58, 59, 60, 86, 134.
c1: 58, 59, 60, 86, 134.
c2: 58, 59, 60, 86, 134.
c3: 58, 59, 60, 86, 134.
d: 69, 75, 77, 78, 80.
decr: 4, 32, 33, 42, 49, 66, 80, 87, 92, 102, 121, 123*, 130*, 140, 141, 142.
delta: 114, 115*.
depth: 44, 74, 104, 115*, 130*, 131.
design_size: 67, 70, 88, 133.
design_size_code: 44, 47, 85.
design_size_loc: 70, 133.
design_units: 67, 70, 89, 134, 136*.
design_units_code: 44, 47, 85.

dict_ptr: 36, 37, 45.
dictionary: 36, 42, 45.
do_nothing: 4, 96, 111, 122, 124* 148*
double_check_ext: 126.
double_check_lig: 126.
double_check_rep: 126.
double_check_tail: 126.
ec: 129, 130* 131, 134, 135, 140.
endif: 123*
enter_name: 45, 46.
eof: 28.
eoln: 28.
equiv: 44, 45, 47, 49.
err_print: 27* 29, 32, 33, 34, 35, 49, 75, 82, 83, 87, 88, 89, 90, 96, 99, 100, 101, 102, 105.
eval: 123* 124*
excess: 78, 79* 80.
existence_tail: 111.
ext_tag: 72, 96, 105, 111.
exten: 67, 105, 106, 112, 126, 143.
extend_filename: 148*
Extra right parenthesis: 82.
extra_loc_needed: 138, 139, 141, 142.
f: 123* 124*
fabs: 136*
face_code: 44, 47, 85.
face_loc: 70, 85.
false: 24, 27* 28, 42, 62, 70, 90, 94, 97, 99, 100, 110, 121, 130* 139, 141, 153*
family_code: 44, 47, 85.
family_loc: 70, 85.
File ended unexpectedly...: 33.
fill_buffer: 28, 29, 31, 32, 33.
finish_inner_property_list: 92, 94, 103* 105.
finish_the_property: 35, 84, 92, 93, 95, 104, 106.
first_ord: 18* 20.
fix_word: 61, 62, 67, 71, 72, 75, 76, 77, 78, 80, 114, 129, 136*
flag: 149* 150* 151* 154*
flush_error: 34, 84, 93, 95, 104, 106.
font_dimen_code: 44, 47, 85.
forward: 124*
four_bytes: 57, 58, 59, 67.
fprint_real: 2*
fraction_digits: 65, 66.
g: 146.
get_byte: 51, 85, 91, 93, 97, 100, 101, 102, 103* 104, 106.
get_fix: 62, 88, 89, 93, 102, 104.
get_four_bytes: 59, 86.
get_keyword_char: 31, 49.
get_name: 49, 84, 93, 95, 104, 106.
get_next: 32, 34, 35, 49, 51, 52, 53, 54, 55, 56, 59, 60, 62, 63, 64, 66, 82, 87, 90, 92, 94, 97, 103* 105.
getopt: 148*
getopt_long_only: 148*
getopt_return_val: 148*
getopt_struct: 148*
good_indent: 21, 22, 29.
h: 40, 75, 77, 78, 80, 118, 123* 124*
has_arg: 149* 150* 151* 154*
hash: 118, 119, 121, 123* 124* 125.
hash_input: 120, 121.
hash_list: 118, 121, 125, 146.
hash_prime: 39, 40, 41, 42, 43.
hash_ptr: 118, 119, 121, 125.
hash_size: 3* 118, 119, 121, 123* 125, 146.
header: 9.
HEADER indices...: 91.
header_bytes: 67, 69, 70, 85, 86, 87, 91, 132, 133, 134.
header_code: 44, 47, 85.
header_index: 67, 68, 69, 86, 87.
header_ptr: 67, 70, 91, 130* 133.
height: 44, 71, 74, 104, 115* 130* 131.
hh: 125, 146.
I had to round...: 115*
ifdef: 123*
Illegal character...: 32.
Illegal digit: 60.
Illegal face code...: 56.
incr: 4, 28, 29, 31, 32, 33, 45, 49, 56, 66, 75, 77, 80, 82, 87, 91, 92, 93, 101, 102, 105, 108, 116, 121, 130* 140, 141, 149* 150* 151*
indent: 21, 22, 29.
index: 79* 80, 135.
index_var: 79*
indx: 68, 118, 121, 123* 124*
Infinite ligature loop...: 125.
initialize: 2* 147*
input_has_ended: 23, 24, 28, 33, 82.
int_part: 62.
integer: 19, 21, 33, 51, 59, 61, 62, 65, 77, 78, 121, 123* 148*
invalid_code: 20, 32.
italic: 44, 71, 74, 104, 115* 130* 131, 132, 137.
j: 42, 62, 132.
Junk after property value...: 35.
junk_error: 82, 83, 92, 94, 103* 105.
k: 19, 27* 42, 45, 78, 87.
kern: 67, 98, 102, 142, 146.
kern_flag: 98, 102, 120, 122, 126.
key: 121, 123*

kpse_set_program_name: 2*
KRN character examined...: 120.
krn_code: 44, 47, 95.
krn_ptr: 98, 102, 142, 146.
l: 33, 77, 80.
label_code: 44, 47, 95.
label_ptr: 138, 140, 141, 142.
label_table: 138, 140, 141, 142.
last_ord: 18* 20.
left_ln: 23, 24, 27* 28.
left_z: 117* 122, 124*
level: 21, 22, 29, 33, 49, 92, 94, 103* 105.
lf: 129, 130* 131.
lh: 129, 130* 131.
LIG character examined...: 120.
LIG character generated...: 120.
lig_code: 44, 47, 95, 101.
lig_exam: 120.
lig_gen: 120.
lig_kern: 67, 99, 100, 101, 102, 116, 118, 120, 122, 126, 139, 142, 146.
lig_ptr: 118, 120, 126, 142, 146.
lig_table_code: 44, 47, 85.
lig_tag: 72, 96, 97, 111, 125, 140.
lig_z: 118, 121, 124*
limit: 23, 24, 27* 28, 29, 31, 32, 33.
line: 21, 22, 27* 28.
link: 71, 72, 74, 75, 77, 80, 137.
list_tag: 72, 96, 104, 111, 113.
lk_offset: 130* 131, 138, 139, 141, 142.
lk_step_ended: 94, 97, 98, 99, 100, 101, 102.
load10: 46, 47, 48.
load11: 46, 47.
load12: 46, 47.
load13: 46, 48.
load14: 46.
load15: 46.
load16: 46, 47.
load17: 46.
load18: 46.
load19: 46.
load20: 46, 48.
load3: 46, 47.
load4: 46, 47, 48.
load5: 46, 47, 48.
load6: 46, 47, 48.
load7: 46, 47, 48.
load8: 46, 47.
load9: 46, 47.
loc: 23, 24, 27* 28, 29, 31, 32, 33, 49, 52, 82, 92.
long_options: 148* 149* 150* 151* 154*
longest_name: 38, 42, 45, 49.
lookup: 42, 45, 49.
m: 77, 80, 136*
max_header_bytes: 3* 9, 68, 91, 132.
max_kerns: 3* 67, 98, 102, 146.
max_letters: 36, 42.
max_lig_steps: 3* 67, 100, 101, 102, 118, 146.
max_name_index: 36, 38, 39, 44.
max_param_words: 3* 11, 67, 93, 132.
Maximum SKIP amount...: 100.
mem_ptr: 72, 74, 75.
mem_size: 71, 75.
memory: 71, 72, 74, 75, 77, 78, 79* 80, 129, 130* 131, 132, 134, 137.
Memory overflow...: 75.
MID piece of character...: 112.
min_cover: 77, 78.
min_nl: 67, 70, 97, 100, 116.
my_name: 1* 2* 148*
n: 136*
n_options: 148*
name: 148* 149* 150* 151* 154*
name_enter: 146, 147*
name_length: 38, 42, 43, 45, 46, 49.
name_ptr: 38, 42, 49.
ne: 67, 70, 105, 106, 126, 130* 131, 143.
negative: 62, 63.
next_d: 76, 77, 78, 88, 89.
next_larger_code: 44, 47, 104.
nhash: 39, 41, 42, 45.
nk: 67, 70, 102, 130* 131, 142.
nl: 67, 70, 97, 99, 100, 101, 102, 110, 116, 120, 125, 126, 130* 131, 139, 142.
no_tag: 72, 74, 96, 111, 113, 125.
not_found: 42, 129, 130*
np: 67, 70, 93, 130* 131, 144.
optind: 148*
option_index: 148*
out: 127* 131, 133, 135, 136* 137, 142, 143, 145.
out_scaled: 136* 137, 142, 144.
out_size: 131, 142.
output: 2*
p: 75, 77, 80, 121, 132.
par_ptr: 132, 144.
param: 67, 93, 144, 145.
param_enter: 146.
PARAMETER index must not...: 93.
parameter_code: 44, 47, 48, 93.
parse_arguments: 2* 148*
pending: 117* 118, 124*
perfect: 25* 26* 27* 147*
pl_file: 2* 5, 6* 28.
pl_name: 6* 148* 155*

PLtoTF: 2*
PLTOTF_HELP: 148*
pointer: 71, 72, 75, 77, 78, 79*, 80, 132.
print: 2*, 6*, 27*, 107, 108, 111, 113, 115*, 125, 126, 136*
print_ln: 2*, 6*, 27*, 75, 108, 110, 111, 113, 115*, 125, 126, 130*, 136*, 147*
print_octal: 107, 108, 111, 113, 125, 126.
print_real: 2*, 115*, 136*
print_version_and_exit: 148*
putbyte: 127*
q: 59, 80, 132.
r: 59.
read: 28.
read_BCPL: 85, 87.
read_char_info: 85, 146.
read_four_bytes: 85, 86, 91.
read_input: 146, 147*
read_lig_kern: 85, 146.
read_ln: 28.
Real constants must be...: 62, 64.
REP piece of character...: 112.
reset: 6*
rewritebin: 16*
right_ln: 23, 24, 27*, 28, 31.
right_z: 117*, 122, 124*
round: 134, 136*
round_message: 115*
rr: 138, 140, 141, 142.
set_indices: 80, 115*
seven_bit_safe_flag: 67, 70, 90, 110.
seven_bit_safe_flag_code: 44, 47, 85.
seven_flag_loc: 70, 133.
seven_unsafe: 109, 110, 111, 120, 133.
shorten: 78, 115*
show_error_context: 27*
simple: 117*, 118, 121, 122, 124*, 125.
SKIP must follow LIG or KRN: 100.
skip_code: 44, 47, 95.
skip_error: 34, 51, 52, 53, 54, 55, 56, 59, 60, 62, 64, 91.
skip_to_end_of_item: 33, 34, 35, 84, 92, 93, 94, 95, 103*, 104, 105, 106.
skip_to_paren: 34, 83, 90, 97.
Sorry, I don't know...: 49.
Sorry, I haven't room...: 125.
Sorry, LIGTABLE too long...: 100, 101, 102.
Sorry, the maximum hex...: 60.
Sorry, the maximum octal...: 60.
Sorry, too many different kerns...: 102.
sort_in: 75, 103*, 104, 111, 126.
sort_ptr: 138, 140, 141, 142.
start: 36, 37, 38, 39, 42, 44, 45.
start_ptr: 36, 37, 45.
stderr: 2*, 148*
STOP must follow LIG or KRN: 99.
stop_code: 44, 47, 95.
stop_flag: 98, 99, 116, 120.
strcmp: 148*
String is too long...: 87.
system dependencies: 2*, 16*, 18*, 28.
t: 51, 121, 138.
tail: 46.
temp_width: 129, 134.
text: 5.
tfm_file: 2*, 15, 16*, 127*
tfm_name: 16*, 148*, 155*
The character NEXTLARGER...: 111.
The design size must...: 88.
The flag value should be...: 90.
The font is not...safe: 110.
The number of units...: 89.
The relative dimension...: 136*
There's junk here...: 83.
This character already...: 96.
This HEADER index is too big...: 91.
This PARAMETER index is too big...: 93.
This property name doesn't belong...: 84, 93, 95, 104, 106.
This value shouldn't...: 53, 54, 55.
TOP piece of character...: 112.
true: 24, 26*, 28, 42, 85, 90, 101, 102, 111, 120, 121, 130*, 139.
tt: 118, 125.
t1: 46.
t10: 46.
t11: 46.
t12: 46.
t13: 46.
t14: 46.
t15: 46.
t16: 46.
t17: 46.
t18: 46.
t19: 46.
t2: 46.
t20: 46.
t3: 46.
t4: 46.
t5: 46.
t6: 46.
t7: 46.
t8: 46.
t9: 46.

uexit: 130*, 147*
unity: 61, 62, 70, 88, 134, 136*
`UNSPECIFIED`: 70.
`Unused KRN step...`: 126.
`Unused LIG step...`: 126.
`Unused VARCHAR...`: 126.
usage: 148*
usage_help: 148*
val: 149*, 150*, 151*, 154*
var_char_code: 44, 47, 104, 106.
verbose: 6*, 103*, 147*, 151*, 152*, 153*
version_string: 6*
`Warning: Inconsistent indentation...`: 29.
`Warning: Indented line...`: 29.
width: 44, 71, 74, 75, 103*, 104, 111, 115*, 126, 129, 130*, 131, 132, 137.
write: 2*
write_ln: 2*, 148*
x: 123*, 124*
x_lig_cycle: 118, 124*, 125.
xord: 18*, 20, 28, 31, 32, 52.
y: 121, 123*, 124*
y_lig_cycle: 118, 119, 124*, 125.
`You need "C" or "D" ...here`: 51.
zz: 121, 122.

⟨ Check for infinite ligature loops 125 ⟩ Used in section 110.
⟨ Check ligature program of c 120 ⟩ Used in sections 110 and 111.
⟨ Check the pieces of *exten*[c] 112 ⟩ Used in section 111.
⟨ Compute the check sum 134 ⟩ Used in section 133.
⟨ Compute the command parameters y, cc, and zz 122 ⟩ Used in section 121.
⟨ Compute the hash code, *cur_hash*, for *cur_name* 43 ⟩ Used in section 42.
⟨ Compute the ligature/kern program offset 139 ⟩ Used in section 130*.
⟨ Compute the twelve subfile sizes 130* ⟩ Used in section 128.
⟨ Constants in the outer block 3* ⟩ Used in section 2*.
⟨ Correct and check the information 110 ⟩ Used in section 146.
⟨ Define the option table 149*, 150*, 151*, 154* ⟩ Used in section 148*.
⟨ Define *parse_arguments* 148* ⟩ Used in section 2*.
⟨ Do the output 128 ⟩ Used in section 147*.
⟨ Doublecheck the lig/kern commands and the extensible recipes 126 ⟩ Used in section 110.
⟨ Enter all of the names and their equivalents, except the parameter names 47 ⟩ Used in section 146.
⟨ Enter the parameter names 48 ⟩ Used in section 146.
⟨ Find the minimum *lk_offset* and adjust all remainders 141 ⟩ Used in section 139.
⟨ For all characters g generated by c, make sure that *char_wd*[g] is nonzero, and set *seven_unsafe* if $c < 128 \le g$ 111 ⟩ Used in section 110.
⟨ Globals in the outer block 5, 15, 18*, 21, 23, 25*, 30, 36, 38, 39, 44, 58, 65, 67, 72, 76, 79*, 81, 98, 109, 114, 118, 129, 132, 138, 152*, 155* ⟩ Used in section 2*.
⟨ Initialize the option variables 153* ⟩ Used in section 148*.
⟨ Insert all labels into *label_table* 140 ⟩ Used in section 139.
⟨ Local variables for initialization 19, 40, 69, 73 ⟩ Used in section 2*.
⟨ Make sure that c is not the largest element of a charlist cycle 113 ⟩ Used in section 110.
⟨ Make sure the ligature/kerning program ends appropriately 116 ⟩ Used in section 110.
⟨ Multiply by 10, add *cur_char* − "0", and *get_next* 64 ⟩ Used in section 62.
⟨ Multiply by r, add *cur_char* − "0", and *get_next* 60 ⟩ Used in section 59.
⟨ Output the character info 135 ⟩ Used in section 128.
⟨ Output the dimensions themselves 137 ⟩ Used in section 128.
⟨ Output the extensible character recipes 143 ⟩ Used in section 128.
⟨ Output the header block 133 ⟩ Used in section 128.
⟨ Output the ligature/kern program 142 ⟩ Used in section 128.
⟨ Output the parameters 144 ⟩ Used in section 128.
⟨ Output the slant (*param*[1]) without scaling 145 ⟩ Used in section 144.
⟨ Output the twelve subfile sizes 131 ⟩ Used in section 128.
⟨ Print c in octal notation 108 ⟩ Used in section 103*.
⟨ Put the width, height, depth, and italic lists into final form 115* ⟩ Used in section 110.
⟨ Read a character property 104 ⟩ Used in section 103*.
⟨ Read a font property value 84 ⟩ Used in section 82.
⟨ Read a kerning step 102 ⟩ Used in section 95.
⟨ Read a label step 97 ⟩ Used in section 95.
⟨ Read a ligature step 101 ⟩ Used in section 95.
⟨ Read a ligature/kern command 95 ⟩ Used in section 94.
⟨ Read a parameter value 93 ⟩ Used in section 92.
⟨ Read a skip step 100 ⟩ Used in section 95.
⟨ Read a stop step 99 ⟩ Used in section 95.
⟨ Read all the input 82 ⟩ Used in section 146.
⟨ Read an extensible piece 106 ⟩ Used in section 105.
⟨ Read an extensible recipe for c 105 ⟩ Used in section 104.
⟨ Read an indexed header word 91 ⟩ Used in section 85.
⟨ Read character info list 103* ⟩ Used in section 146.

⟨ Read font parameter list 92 ⟩ Used in section 85.
⟨ Read ligature/kern list 94 ⟩ Used in section 146.
⟨ Read the design size 88 ⟩ Used in section 85.
⟨ Read the design units 89 ⟩ Used in section 85.
⟨ Read the font property value specified by *cur_code* 85 ⟩ Used in section 84.
⟨ Read the seven-bit-safe flag 90 ⟩ Used in section 85.
⟨ Scan a face code 56 ⟩ Used in section 51.
⟨ Scan a small decimal number 53 ⟩ Used in section 51.
⟨ Scan a small hexadecimal number 55 ⟩ Used in section 51.
⟨ Scan a small octal number 54 ⟩ Used in section 51.
⟨ Scan an ASCII character code 52 ⟩ Used in section 51.
⟨ Scan the blanks and/or signs after the type code 63 ⟩ Used in section 62.
⟨ Scan the fraction part and put it in *acc* 66 ⟩ Used in section 62.
⟨ Set initial values 6*, 16*, 20, 22, 24, 26*, 37, 41, 70, 74, 119 ⟩ Used in section 2*.
⟨ Set *loc* to the number of leading blanks in the buffer, and check the indentation 29 ⟩ Used in section 28.
⟨ Types in the outer block 17, 57, 61, 68, 71 ⟩ Used in section 2*.