The PLtoTF processor

(Version 3.6, January 2014)

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.

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 kth 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* ≠ 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.

⟨ 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*.