The TFtoPL processor

(Version 3.3, 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 TFtoPL utility program converts TEX font metric ("TFM") files into equivalent property-list ("PL") files. It also makes a thorough check of the given TFM file, using essentially the same algorithm as TEX. Thus if TEX complains that a TFM file is "bad," this program will pinpoint the source or sources of badness. A PL file output by this program can be edited with a normal text editor, and the result can be converted back to TFM format using the companion program PLtoTF.

The first TFtoPL 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 TFtoPL gets modified.

```
define my_name ≡ 'tftopl'
define banner ≡ 'This␣is␣TFtoPL,␣Version␣3.3'  { printed when the program starts }
```

2* This program is written entirely in standard Pascal, except that it occasionally has lower case letters in strings that are output. Such letters can be converted to upper case if necessary. The input is read from *tfm_file*, and the output is written on *pl_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,#)
  { Tangle doesn't recognize @ when it's right after the =. }
@\␣@define␣var␣tfm;@\
program TFtoPL(tfm_file,pl_file,output);
const ⟨ Constants in the outer block 4* ⟩
type ⟨ Types in the outer block 18* ⟩
var ⟨ Globals in the outer block 6 ⟩
  ⟨ Define parse_arguments 100* ⟩
procedure initialize;  { this procedure gets things started properly }
  begin kpse_set_program_name(argv[0],my_name); kpse_init_prog('TFTOPL',0,nil,nil);
    { We xrealloc when we know how big the file is. The 1000 comes from the negative lower bound. }
  tfm_file_array ← xmalloc_array(byte,1002); parse_arguments; ⟨ Set initial values 7* ⟩
  end;
```

3* If the program has to stop prematurely, it goes to the '*final_end*'.

```
define final_end = 9999  { label for the end of it all }
```

4* The following parameters can be changed at compile time to extend or reduce TFtoPL's capacity.

```
⟨ Constants in the outer block 4* ⟩ ≡
  lig_size = 32510;  { maximum length of lig_kern program, in words (< 2^15) }
  hash_size = 32579;
    { preferably a prime number, a bit larger than the number of character pairs in lig/kern steps }
```

See also section 108*.

This code is used in section 2*.

7* On some systems you may have to do something special to read a packed file of bytes. With C under Unix, we just open the file by name and read characters from it.

```
⟨ Set initial values 7* ⟩ ≡
  tfm_file ← kpse_open_file(tfm_name, kpse_tfm_format);
  if verbose then
    begin print(banner); print_ln(version_string);
    end;
```

See also sections 17*, 28*, 33, 46, and 64.

This code is used in section 2*.

17* If an explicit filename isn't given, we write to *stdout*.

```
⟨ Set initial values 7* ⟩ +≡
  if optind + 1 = argc then
    begin pl_file ← stdout;
    end
  else begin pl_name ← extend_filename(cmdline(optind + 1), 'pl'); rewrite(pl_file, pl_name);
    end;
```

18.* Unpacked representation. The first thing `TFtoPL` does is read the entire *tfm_file* into an array of bytes, $tfm[0 \,..\, (4 * lf - 1)]$.

define *index* ≡ *index_type*

⟨ Types in the outer block 18* ⟩ ≡
```
byte = 0 .. 255;   { unsigned eight-bit quantity }
index = integer;   { address of a byte in tfm }
```
See also section 107*.

This code is used in section 2*.

19.* ⟨ Globals in the outer block 6 ⟩ +≡
```
{ Kludge here to define tfm as a macro which takes care of the negative lower bound. We've defined
  tfm for the benefit of web2c above. }
#define tfm (tfmfilearray + 1001);
@\tfm_file_array: ↑byte;   { the input data all goes here }
{ the negative addresses avoid range checks for invalid characters }
```

20.* The input may, of course, be all screwed up and not a `TFM` file at all. So we begin cautiously.

define *abort*(#) ≡
```
begin print_ln(#);
print_ln('Sorry, but I can't go on; are you sure this is a TFM?'); goto final_end;
end
```

⟨ Read the whole input file 20* ⟩ ≡
```
read(tfm_file, tfm[0]);
if tfm[0] > 127 then abort('The first byte of the input file exceeds 127!');
if eof(tfm_file) then abort('The input file is only one byte long!');
read(tfm_file, tfm[1]); lf ← tfm[0] * '400 + tfm[1];
if lf = 0 then abort('The file claims to have length zero, but that's impossible!');
tfm_file_array ← xrealloc_array(tfm_file_array, byte, 4 * lf + 1000);
for tfm_ptr ← 2 to 4 * lf - 1 do
  begin if eof(tfm_file) then abort('The file has fewer bytes than it claims!');
  read(tfm_file, tfm[tfm_ptr]);
  end;
if ¬eof(tfm_file) then
  begin print_ln('There's some extra junk at the end of the TFM file,');
  print_ln('but I'll proceed as if it weren't there.');
  end
```

This code is used in section 96.

27:* In order to stick to standard Pascal, we use three strings called *ASCII_04*, *ASCII_10*, and *ASCII_14*, in terms of which we can do the appropriate conversion of ASCII codes. Three other little strings are used to produce *face* codes like `MIE`.

⟨ Globals in the outer block 6 ⟩ +≡
ASCII_04, *ASCII_10*, *ASCII_14*: *const_c_string*; { strings for output in the user's external character set }
ASCII_all: **packed array** [0 .. 256] **of** *char*;
MBL_string, *RI_string*, *RCE_string*: *const_c_string*; { handy string constants for *face* codes }

28:* ⟨ Set initial values 7* ⟩ +≡
ASCII_04 ← ´␣␣!"#$%&´´()*+,-./0123456789:;<=>?´;
ASCII_10 ← ´␣@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_´;
ASCII_14 ← ´␣`abcdefghijklmnopqrstuvwxyz{|}~␣´;
strcpy(*ASCII_all*, *ASCII_04*); *strcat*(*ASCII_all*, ´@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_´);
strcat(*ASCII_all*, ´`abcdefghijklmnopqrstuvwxyz{|}~´);
MBL_string ← ´␣MBL´; *RI_string* ← ´␣RI␣´; *RCE_string* ← ´␣RCE´;

38:* The property value may be a character, which is output in octal unless it is a letter or a digit. This procedure is the only place where a lowercase letter will be output to the `PL` file.

procedure *out_char*(*c* : *byte*); { outputs a character }
  begin if (*font_type* > *vanilla*) ∨ (*charcode_format* = *charcode_octal*) **then**
    begin *tfm*[0] ← *c*; *out_octal*(0, 1)
    end
  else if (*charcode_format* = *charcode_ascii*) ∧ (*c* > "␣") ∧ (*c* ≤ "~") ∧ (*c* ≠ "(") ∧ (*c* ≠ ")") **then**
      out(´␣C␣´, *ASCII_all*[*c* − "␣" + 1]) { default case, use `C` only for letters and digits }
    else if (*c* ≥ "0") ∧ (*c* ≤ "9") **then** *out*(´␣C␣´, *c* − "0" : 1)
      else if (*c* ≥ "A") ∧ (*c* ≤ "Z") **then** *out*(´␣C␣´, *ASCII_10*[*c* − "A" + 2])
        else if (*c* ≥ "a") ∧ (*c* ≤ "z") **then** *out*(´␣C␣´, *ASCII_14*[*c* − "a" + 2])
          else begin *tfm*[0] ← *c*; *out_octal*(0, 1);
            end;
  end;

39:* The property value might be a "face" byte, which is output in the curious code mentioned earlier, provided that it is less than 18.

procedure *out_face*(*k* : *index*); { outputs a *face* }
  var *s*: 0 .. 1; { the slope }
    b: 0 .. 8; { the weight and expansion }
  begin if *tfm*[*k*] ≥ 18 **then** *out_octal*(*k*, 1)
  else begin *out*(´␣F␣´); { specify face-code format }
    s ← *tfm*[*k*] **mod** 2; *b* ← *tfm*[*k*] **div** 2; *put_byte*(*MBL_string*[1 + (*b* **mod** 3)], *pl_file*);
    put_byte(*RI_string*[1 + *s*], *pl_file*); *put_byte*(*RCE_string*[1 + (*b* **div** 3)], *pl_file*);
    end;
  end;

78:* The last thing on TFtoPL's agenda is to go through the list of *char_info* and spew out the information about each individual character.

```
⟨ Do the characters 78* ⟩ ≡
  sort_ptr ← 0;  { this will suppress 'STOP' lines in ligature comments }
  for c ← bc to ec do
    if width_index(c) > 0 then
      begin if chars_on_line = 8 then
        begin print_ln('␣'); chars_on_line ← 1;
        end
      else begin if chars_on_line > 0 then print('␣');
        if verbose then incr(chars_on_line);
        end;
      if verbose then print_octal(c);  { progress report }
      left; out('CHARACTER'); out_char(c); out_ln; ⟨ Output the character's width 79 ⟩;
      if height_index(c) > 0 then ⟨ Output the character's height 80 ⟩;
      if depth_index(c) > 0 then ⟨ Output the character's depth 81 ⟩;
      if italic_index(c) > 0 then ⟨ Output the italic correction 82 ⟩;
      case tag(c) of
      no_tag: do_nothing;
      lig_tag: ⟨ Output the applicable part of the ligature/kern program as a comment 83 ⟩;
      list_tag: ⟨ Output the character link unless there is a problem 84 ⟩;
      ext_tag: ⟨ Output an extensible character recipe 85 ⟩;
      end;  { there are no other cases }
      right;
      end
```

This code is used in section 98.

89:* To detect such loops, `TFtoPL` attempts to evaluate the function $f(x,y)$ for all character pairs x and y, where f is defined as follows: If the current character is x and the next character is y, we say the "cursor" is between x and y; when the cursor first moves past y, the character immediately to its left is $f(x,y)$. This function is defined if and only if no infinite loop is generated when the cursor is between x and y.

The function $f(x,y)$ can be defined recursively. It turns out that all pairs (x,y) belong to one of five classes. The simplest class has $f(x,y)=y$; this happens if there's no ligature between x and y, or in the cases `LIG/>` and `/LIG/>>`. Another simple class arises when there's a `LIG` or `/LIG>` between x and y, generating the character z; then $f(x,y)=z$. Otherwise we always have $f(x,y)$ equal to either $f(x,z)$ or $f(z,y)$ or $f(f(x,z),y)$, where z is the inserted ligature character.

The first two of these classes can be merged; we can also consider (x,y) to belong to the simple class when $f(x,y)$ has been evaluated. For technical reasons we allow x to be 256 (for the boundary character at the left) or 257 (in cases when an error has been detected).

For each pair (x,y) having a ligature program step, we store (x,y) in a hash table from which the values z and *class* can be read.

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* $\equiv$ *class_var*

⟨ Globals in the outer block 6 ⟩ +≡
hash: **array** [0 .. *hash_size*] **of** 0 .. 66048; { $256x+y+1$ for $x\le 257$ and $y\le 255$ }
class: **array** [0 .. *hash_size*] **of** *simple* .. *pending*;
lig_z: **array** [0 .. *hash_size*] **of** 0 .. 257;
hash_ptr: 0 .. *hash_size*; { the number of nonzero entries in *hash* }
hash_list: **array** [0 .. *hash_size*] **of** 0 .. *hash_size*; { list of those nonzero entries }
h, *hh*: 0 .. *hash_size*; { indices into the hash table }
x_lig_cycle, *y_lig_cycle*: 0 .. 256; { problematic ligature pair }

90* ⟨ Check for ligature cycles 90* ⟩ ≡

```
hash_ptr ← 0; y_lig_cycle ← 256;
for hh ← 0 to hash_size do hash[hh] ← 0;  { clear the hash table }
for c ← bc to ec do
  if tag(c) = lig_tag then
    begin i ← remainder(c);
    if tfm[lig_step(i)] > stop_flag then i ← 256 * tfm[lig_step(i) + 2] + tfm[lig_step(i) + 3];
    ⟨ Enter data for character c starting at location i in the hash table 91 ⟩;
    end;
if bchar_label < nl then
  begin c ← 256; i ← bchar_label;
  ⟨ Enter data for character c starting at location i in the hash table 91 ⟩;
  end;
if hash_ptr = hash_size then
  begin print_ln('Sorry, I haven''t room for so many ligature/kern pairs!'); uexit(1); ;
  end;
for hh ← 1 to hash_ptr do
  begin r ← hash_list[hh];
  if class[r] > simple then  { make sure f is defined }
    r ← f_fn(r, (hash[r] − 1) div 256, (hash[r] − 1) mod 256);
  end;
if y_lig_cycle < 256 then
  begin print('Infinite ligature loop starting with '');
  if x_lig_cycle = 256 then print('boundary') else print_octal(x_lig_cycle);
  print(' and '); print_octal(y_lig_cycle); print_ln('!');
  out('(INFINITE LIGATURE LOOP MUST BE BROKEN!)'); uexit(1);
  end
```

This code is used in section 66.

94* Evaluation of $f(x, y)$ is handled by two mutually recursive procedures. Kind of a neat algorithm, generalizing a depth-first search.

```
ifdef('notdef')
function f_fn(h, x, y : index): index;
  begin end;
  { compute f for arguments known to be in hash[h] }
endif('notdef')
function eval(x, y : index): index;  { 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_fn(h, x, y);
  end;
```

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

```
function f_fn(h,x,y : index): index;
  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_fn ← lig_z[h];
  end;
```

99.* Here is where `TFtoPL` begins and ends.

```
begin initialize;
if ¬organize then uexit(1);
do_simple_things;
⟨ Do the ligatures and kerns 66 ⟩;
⟨ Check the extensible recipes 87 ⟩;
do_characters;
if verbose then print_ln('.');
if level ≠ 0 then print_ln('This␣program␣isn''t␣working!');
if ¬perfect then
  begin out('(COMMENT␣THE␣TFM␣FILE␣WAS␣BAD,␣SO␣THE␣DATA␣HAS␣BEEN␣CHANGED!)');
  write_ln(pl_file);
  end;
end.
```

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

```
define argument_is(#) ≡ (strcmp(long_options[option_index].name, #) = 0)
⟨ Define parse_arguments 100* ⟩ ≡
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 105* ⟩;
  ⟨ Define the option table 101* ⟩;
  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(TFTOPL_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(´charcode-format´) then
              begin if strcmp(optarg, ´ascii´) = 0 then charcode_format ← charcode_ascii
              else if strcmp(optarg, ´octal´) = 0 then charcode_format ← charcode_octal
                else print_ln(´Bad␣character␣code␣format␣´, stringcast(optarg), ´.´);
              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. }
  if (optind + 1 ≠ argc) ∧ (optind + 2 ≠ argc) then
    begin print_ln(my_name, ´:␣Need␣one␣or␣two␣file␣arguments.´); usage(my_name);
    end;
  tfm_name ← cmdline(optind);
  end;
```

This code is used in section 2*.

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

```
⟨ Define the option table 101* ⟩ ≡
  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 102*, 103*, 106*, and 111*.

This code is used in section 100*.

102.* Another of the standard options.

```
⟨ Define the option table 101* ⟩ +≡
  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);
```

103.* Print progress information?

⟨ Define the option table 101* ⟩ +≡
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*);

104.* ⟨ Globals in the outer block 6 ⟩ +≡
verbose: *c_int_type*;

105.* ⟨ Initialize the option variables 105* ⟩ ≡
verbose ← *false*;
See also section 110*.
This code is used in section 100*.

106.* This option changes how we output character codes.

⟨ Define the option table 101* ⟩ +≡
long_options[*current_option*].*name* ← ´**charcode-format**´; *long_options*[*current_option*].*has_arg* ← 1;
long_options[*current_option*].*flag* ← 0; *long_options*[*current_option*].*val* ← 0; *incr*(*current_option*);

107.* We use an "enumerated" type to store the information.

⟨ Types in the outer block 18* ⟩ +≡
charcode_format_type = *charcode_ascii* .. *charcode_default*;

108.* ⟨ Constants in the outer block 4* ⟩ +≡
charcode_ascii = 0; *charcode_octal* = 1; *charcode_default* = 2;

109.* ⟨ Globals in the outer block 6 ⟩ +≡
charcode_format: *charcode_format_type*;

110.* It starts off as the default, that is, we output letters and digits as ASCII characters, everything else in octal.

⟨ Initialize the option variables 105* ⟩ +≡
charcode_format ← *charcode_default*;

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

⟨ Define the option table 101* ⟩ +≡
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;

112.* Global filenames.

⟨ Globals in the outer block 6 ⟩ +≡
tfm_name, *pl_name*: *const_c_string*;

113* 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, 4, 7, 17, 18, 19, 20, 27, 28, 38, 39, 78, 89, 90, 94, 95, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113.