The TFtoPL processor

(Version 3.3, January 2014)

	Section	Page
Introduction	1	202
Font metric data	6	203
Unpacked representation	18	209
Basic output subroutines	26	212
Doing it	44	216
Checking for ligature loops	88	228
The main program	96	232
System-dependent changes	100	233
Index	101	234

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 *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*(#)
define *print_ln*(#) ≡ *write_ln*(#)

program *TFtoPL*(*tfm_file*, *pl_file*, *output*);
label ⟨ Labels in the outer block 3 ⟩
const ⟨ Constants in the outer block 4 ⟩
type ⟨ Types in the outer block 18 ⟩
var ⟨ Globals in the outer block 6 ⟩
procedure *initialize*; { this procedure gets things started properly }
begin *print_ln*(*banner*);
⟨ 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 }

⟨ Labels in the outer block 3 ⟩ ≡
final_end;

This code is used in section 2.

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

⟨ Constants in the outer block 4 ⟩ ≡
tfm_size = 30000; { maximum length of *tfm* data, in bytes }
lig_size = 5000; { maximum length of *lig_kern* program, in words }
hash_size = 5003;
{ preferably a prime number, a bit larger than the number of character pairs in lig/kern steps }

This code is used in section 2.

5. 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 *do_nothing* ≡ { empty statement }

6. Font metric data. The idea behind TFM files is that typesetting routines like TEX need a compact way to store the relevant information about several dozen fonts, and computer centers need a compact way to store the relevant information about several hundred fonts. TFM files are compact, and most of the information they contain is highly relevant, so they provide a solution to the problem.

The information in a TFM file appears in a sequence of 8-bit bytes. Since the number of bytes is always a multiple of 4, we could also regard the file as a sequence of 32-bit words; but TEX uses the byte interpretation, and so does TFtoPL. Note that the bytes are considered to be unsigned numbers.

⟨ Globals in the outer block 6 ⟩ ≡
tfm_file: **packed file of** 0 .. 255;

See also sections 8, 16, 19, 22, 25, 27, 29, 32, 45, 47, 63, 65, and 89.

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. For example, the following code didn't work when it was first tried at Stanford, because packed files have to be opened with a special switch setting on the Pascal that was used.

⟨ Set initial values 7 ⟩ ≡
reset(*tfm_file*);

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

This code is used in section 2.

8. The first 24 bytes (6 words) of a TFM file contain twelve 16-bit integers that give the lengths of the various subsequent portions of the file. These twelve integers are, in order:

lf = length of the entire file, in words;
lh = length of the header data, in words;
bc = smallest character code in the font;
ec = largest character code in the font;
nw = number of words in the width table;
nh = number of words in the height table;
nd = number of words in the depth table;
ni = number of words in the italic correction table;
nl = number of words in the lig/kern table;
nk = number of words in the kern table;
ne = number of words in the extensible character table;
np = number of font parameter words.

They are all nonnegative and less than 2^{15}. We must have $bc - 1 \le ec \le 255$, $ne \le 256$, and

$$lf = 6 + lh + (ec - bc + 1) + nw + nh + nd + ni + nl + nk + ne + np.$$

Note that a font may contain as many as 256 characters (if $bc = 0$ and $ec = 255$), and as few as 0 characters (if $bc = ec + 1$).

Incidentally, when two or more 8-bit bytes are combined to form an integer of 16 or more bits, the most significant bytes appear first in the file. This is called BigEndian order.

⟨ Globals in the outer block 6 ⟩ +≡
$lf, lh, bc, ec, nw, nh, nd, ni, nl, nk, ne, np$: 0 .. ′77777; { subfile sizes }

9. The rest of the TFM file may be regarded as a sequence of ten data arrays having the informal specification

$$\begin{aligned}
\mathit{header} &: \textbf{array}\ [0\,..\,\mathit{lh}-1]\ \textbf{of}\ \mathit{stuff}\\
\mathit{char_info} &: \textbf{array}\ [\mathit{bc}\,..\,\mathit{ec}]\ \textbf{of}\ \mathit{char_info_word}\\
\mathit{width} &: \textbf{array}\ [0\,..\,\mathit{nw}-1]\ \textbf{of}\ \mathit{fix_word}\\
\mathit{height} &: \textbf{array}\ [0\,..\,\mathit{nh}-1]\ \textbf{of}\ \mathit{fix_word}\\
\mathit{depth} &: \textbf{array}\ [0\,..\,\mathit{nd}-1]\ \textbf{of}\ \mathit{fix_word}\\
\mathit{italic} &: \textbf{array}\ [0\,..\,\mathit{ni}-1]\ \textbf{of}\ \mathit{fix_word}\\
\mathit{lig_kern} &: \textbf{array}\ [0\,..\,\mathit{nl}-1]\ \textbf{of}\ \mathit{lig_kern_command}\\
\mathit{kern} &: \textbf{array}\ [0\,..\,\mathit{nk}-1]\ \textbf{of}\ \mathit{fix_word}\\
\mathit{exten} &: \textbf{array}\ [0\,..\,\mathit{ne}-1]\ \textbf{of}\ \mathit{extensible_recipe}\\
\mathit{param} &: \textbf{array}\ [1\,..\,\mathit{np}]\ \textbf{of}\ \mathit{fix_word}
\end{aligned}$$

The most important data type used here is a *fix_word*, which is a 32-bit representation of a binary fraction. A *fix_word* is a signed quantity, with the two's complement of the entire word used to represent negation. Of the 32 bits in a *fix_word*, exactly 12 are to the left of the binary point; thus, the largest *fix_word* value is $2048 - 2^{-20}$, and the smallest is -2048. We will see below, however, that all but one of the *fix_word* values will lie between -16 and $+16$.

10. The first data array is a block of header information, which contains general facts about the font. The header must contain at least two words, and for `TFM` files to be used with Xerox printing software it must contain at least 18 words, allocated as described below. When different kinds of devices need to be interfaced, it may be necessary to add further words to the header block.

header[0] is a 32-bit check sum that TEX will copy into the `DVI` output file whenever it uses the font. Later on when the `DVI` file is printed, possibly on another computer, the actual font that gets used is supposed to have a check sum that agrees with the one in the `TFM` file used by TEX. In this way, users will be warned about potential incompatibilities. (However, if the check sum is zero in either the font file or the `TFM` file, no check is made.) The actual relation between this check sum and the rest of the `TFM` file is not important; the check sum is simply an identification number with the property that incompatible fonts almost always have distinct check sums.

header[1] is a *fix_word* containing the design size of the font, in units of TEX points (7227 TEX points = 254 cm). This number must be at least 1.0; it is fairly arbitrary, but usually the design size is 10.0 for a "10 point" font, i.e., a font that was designed to look best at a 10-point size, whatever that really means. When a TEX user asks for a font '`at` δ `pt`', the effect is to override the design size and replace it by δ, and to multiply the x and y coordinates of the points in the font image by a factor of δ divided by the design size. *All other dimensions in the* `TFM` *file are fix_word numbers in design-size units.* Thus, for example, the value of *param*[6], one `em` or `\quad`, is often the *fix_word* value $2^{20} = 1.0$, since many fonts have a design size equal to one em. The other dimensions must be less than 16 design-size units in absolute value; thus, *header*[1] and *param*[1] are the only *fix_word* entries in the whole `TFM` file whose first byte might be something besides 0 or 255.

header[2 .. 11], if present, contains 40 bytes that identify the character coding scheme. The first byte, which must be between 0 and 39, is the number of subsequent ASCII bytes actually relevant in this string, which is intended to specify what character-code-to-symbol convention is present in the font. Examples are `ASCII` for standard ASCII, `TeX text` for fonts like `cmr10` and `cmti9`, `TeX math extension` for `cmex10`, `XEROX text` for Xerox fonts, `GRAPHIC` for special-purpose non-alphabetic fonts, `UNSPECIFIED` for the default case when there is no information. Parentheses should not appear in this name. (Such a string is said to be in BCPL format.)

header[12 .. 16], if present, contains 20 bytes that name the font family (e.g., `CMR` or `HELVETICA`), in BCPL format. This field is also known as the "font identifier."

header[17], if present, contains a first byte called the *seven_bit_safe_flag*, then two bytes that are ignored, and a fourth byte called the *face*. If the value of the fourth byte is less than 18, it has the following interpretation as a "weight, slope, and expansion": Add 0 or 2 or 4 (for medium or bold or light) to 0 or 1 (for roman or italic) to 0 or 6 or 12 (for regular or condensed or extended). For example, 13 is 0+1+12, so it represents medium italic extended. A three-letter code (e.g., `MIE`) can be used for such *face* data.

header[18 .. whatever] might also be present; the individual words are simply called *header*[18], *header*[19], etc., at the moment.

11. Next comes the *char_info* array, which contains one *char_info_word* per character. Each *char_info_word* contains six fields packed into four bytes as follows.

first byte: *width_index* (8 bits)
second byte: *height_index* (4 bits) times 16, plus *depth_index* (4 bits)
third byte: *italic_index* (6 bits) times 4, plus *tag* (2 bits)
fourth byte: *remainder* (8 bits)

The actual width of a character is $width[width\_index]$, in design-size units; this is a device for compressing information, since many characters have the same width. Since it is quite common for many characters to have the same height, depth, or italic correction, the TFM format imposes a limit of 16 different heights, 16 different depths, and 64 different italic corrections.

Incidentally, the relation $width[0] = height[0] = depth[0] = italic[0] = 0$ should always hold, so that an index of zero implies a value of zero. The *width_index* should never be zero unless the character does not exist in the font, since a character is valid if and only if it lies between *bc* and *ec* and has a nonzero *width_index*.

12. The *tag* field in a *char_info_word* has four values that explain how to interpret the *remainder* field.

$tag = 0$ (*no_tag*) means that *remainder* is unused.
$tag = 1$ (*lig_tag*) means that this character has a ligature/kerning program starting at $lig\_kern[remainder]$.
$tag = 2$ (*list_tag*) means that this character is part of a chain of characters of ascending sizes, and not the largest in the chain. The *remainder* field gives the character code of the next larger character.
$tag = 3$ (*ext_tag*) means that this character code represents an extensible character, i.e., a character that is built up of smaller pieces so that it can be made arbitrarily large. The pieces are specified in $exten[remainder]$.

define *no_tag* = 0 { vanilla character }
define *lig_tag* = 1 { character has a ligature/kerning program }
define *list_tag* = 2 { character has a successor in a charlist }
define *ext_tag* = 3 { character is extensible }

13. The *lig_kern* array contains instructions in a simple programming language that explains what to do for special letter pairs. Each word is a *lig_kern_command* of four bytes.

first byte: *skip_byte*, indicates that this is the final program step if the byte is 128 or more, otherwise the next step is obtained by skipping this number of intervening steps.

second byte: *next_char*, "if *next_char* follows the current character, then perform the operation and stop, otherwise continue."

third byte: *op_byte*, indicates a ligature step if less than 128, a kern step otherwise.

fourth byte: *remainder*.

In a kern step, an additional space equal to $kern[256 * (op\_byte - 128) + remainder]$ is inserted between the current character and *next_char*. This amount is often negative, so that the characters are brought closer together by kerning; but it might be positive.

There are eight kinds of ligature steps, having *op_byte* codes $4a+2b+c$ where $0 \le a \le b+c$ and $0 \le b, c \le 1$. The character whose code is *remainder* is inserted between the current character and *next_char*; then the current character is deleted if $b = 0$, and *next_char* is deleted if $c = 0$; then we pass over a characters to reach the next current character (which may have a ligature/kerning program of its own).

Notice that if $a = 0$ and $b = 1$, the current character is unchanged; if $a = b$ and $c = 1$, the current character is changed but the next character is unchanged. `TFtoPL` will check to see that infinite loops are avoided.

If the very first instruction of the *lig_kern* array has *skip_byte* = 255, the *next_char* byte is the so-called right boundary character of this font; the value of *next_char* need not lie between *bc* and *ec*. If the very last instruction of the *lig_kern* array has *skip_byte* = 255, there is a special ligature/kerning program for a left boundary character, beginning at location $256 * op\_byte + remainder$. The interpretation is that TEX puts implicit boundary characters before and after each consecutive string of characters from the same font. These implicit characters do not appear in the output, but they can affect ligatures and kerning.

If the very first instruction of a character's *lig_kern* program has *skip_byte* > 128, the program actually begins in location $256 * op\_byte + remainder$. This feature allows access to large *lig_kern* arrays, because the first instruction must otherwise appear in a location ≤ 255.

Any instruction with *skip_byte* > 128 in the *lig_kern* array must have $256 * op\_byte + remainder < nl$. If such an instruction is encountered during normal program execution, it denotes an unconditional halt; no ligature command is performed.

define *stop_flag* = 128 { value indicating '`STOP`' in a lig/kern program }
define *kern_flag* = 128 { op code for a kern step }

14. Extensible characters are specified by an *extensible_recipe*, which consists of four bytes called *top*, *mid*, *bot*, and *rep* (in this order). These bytes are the character codes of individual pieces used to build up a large symbol. If *top*, *mid*, or *bot* are zero, they are not present in the built-up result. For example, an extensible vertical line is like an extensible bracket, except that the top and bottom pieces are missing.

15. The final portion of a TFM file is the *param* array, which is another sequence of *fix_word* values.

param[1] = *slant* is the amount of italic slant, which is used to help position accents. For example, *slant* = .25 means that when you go up one unit, you also go .25 units to the right. The *slant* is a pure number; it's the only *fix_word* other than the design size itself that is not scaled by the design size.

param[2] = *space* is the normal spacing between words in text. Note that character "␣" in the font need not have anything to do with blank spaces.

param[3] = *space_stretch* is the amount of glue stretching between words.

param[4] = *space_shrink* is the amount of glue shrinking between words.

param[5] = *x_height* is the height of letters for which accents don't have to be raised or lowered.

param[6] = *quad* is the size of one em in the font.

param[7] = *extra_space* is the amount added to *param*[2] at the ends of sentences.

When the character coding scheme is TeX math symbols, the font is supposed to have 15 additional parameters called *num1*, *num2*, *num3*, *denom1*, *denom2*, *sup1*, *sup2*, *sup3*, *sub1*, *sub2*, *supdrop*, *subdrop*, *delim1*, *delim2*, and *axis_height*, respectively. When the character coding scheme is TeX math extension, the font is supposed to have six additional parameters called *default_rule_thickness* and *big_op_spacing1* through *big_op_spacing5*.

16. So that is what TFM files hold. The next question is, "What about PL files?" A complete answer to that question appears in the documentation of the companion program, PLtoTF, so it will not be repeated here. Suffice it to say that a PL file is an ordinary Pascal text file, and that the output of TFtoPL uses only a subset of the possible constructions that might appear in a PL file. Furthermore, hardly anybody really wants to look at the formal definition of PL format, because it is almost self-explanatory when you see an example or two.

```
⟨ Globals in the outer block 6 ⟩ +≡
pl_file: text;
```

17. ⟨ Set initial values 7 ⟩ +≡

```
rewrite(pl_file);
```

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

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

This code is used in section 2.

19. ⟨ Globals in the outer block 6 ⟩ +≡

```
tfm: array [−1000 .. tfm_size] of 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!');
  if 4 * lf − 1 > tfm_size then abort('The file is bigger than I can handle!');
  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.

21. After the file has been read successfully, we look at the subfile sizes to see if they check out.

```
define eval_two_bytes(#) ≡
        begin if tfm[tfm_ptr] > 127 then abort('One␣of␣the␣subfile␣sizes␣is␣negative!');
        # ← tfm[tfm_ptr] * '400 + tfm[tfm_ptr + 1]; tfm_ptr ← tfm_ptr + 2;
        end
⟨ Set subfile sizes lh, bc, ..., np 21 ⟩ ≡
  begin tfm_ptr ← 2;
  eval_two_bytes(lh); eval_two_bytes(bc); eval_two_bytes(ec); eval_two_bytes(nw); eval_two_bytes(nh);
  eval_two_bytes(nd); eval_two_bytes(ni); eval_two_bytes(nl); eval_two_bytes(nk); eval_two_bytes(ne);
  eval_two_bytes(np);
  if lh < 2 then abort('The␣header␣length␣is␣only␣', lh : 1, '!');
  if nl > lig_size then abort('The␣lig/kern␣program␣is␣longer␣than␣I␣can␣handle!');
  if (bc > ec + 1) ∨ (ec > 255) then
    abort('The␣character␣code␣range␣', bc : 1, '..', ec : 1, '␣is␣illegal!');
  if (nw = 0) ∨ (nh = 0) ∨ (nd = 0) ∨ (ni = 0) then
    abort('Incomplete␣subfiles␣for␣character␣dimensions!');
  if ne > 256 then abort('There␣are␣', ne : 1, '␣extensible␣recipes!');
  if lf ≠ 6 + lh + (ec − bc + 1) + nw + nh + nd + ni + nl + nk + ne + np then
    abort('Subfile␣sizes␣don''t␣add␣up␣to␣the␣stated␣total!');
  end
```

This code is used in section 96.

22. Once the input data successfully passes these basic checks, TFtoPL believes that it is a TFM file, and the conversion to PL format will take place. Access to the various subfiles is facilitated by computing the following base addresses. For example, the *char_info* for character c will start in location $4 * (char\_base + c)$ of the *tfm* array.

```
⟨ Globals in the outer block 6 ⟩ +≡
char_base, width_base, height_base, depth_base, italic_base, lig_kern_base, kern_base, exten_base, param_base:
      integer;   { base addresses for the subfiles }
```

23.

```
⟨ Compute the base addresses 23 ⟩ ≡
  begin char_base ← 6 + lh − bc; width_base ← char_base + ec + 1; height_base ← width_base + nw;
  depth_base ← height_base + nh; italic_base ← depth_base + nd; lig_kern_base ← italic_base + ni;
  kern_base ← lig_kern_base + nl; exten_base ← kern_base + nk; param_base ← exten_base + ne − 1;
  end
```

This code is used in section 96.

24. Of course we want to define macros that suppress the detail of how the font information is actually encoded. Each word will be referred to by the *tfm* index of its first byte. For example, if c is a character code between *bc* and *ec*, then *tfm*[*char_info*(*c*)] will be the first byte of its *char_info*, i.e., the *width_index*; furthermore *width*(*c*) will point to the *fix_word* for *c*'s width.

define *check_sum* = 24
define *design_size* = *check_sum* + 4
define *scheme* = *design_size* + 4
define *family* = *scheme* + 40
define *random_word* = *family* + 20
define *char_info*(#) ≡ 4 ∗ (*char_base* + #)
define *width_index*(#) ≡ *tfm*[*char_info*(#)]
define *nonexistent*(#) ≡ ((# < *bc*) ∨ (# > *ec*) ∨ (*width_index*(#) = 0))
define *height_index*(#) ≡ (*tfm*[*char_info*(#) + 1] **div** 16)
define *depth_index*(#) ≡ (*tfm*[*char_info*(#) + 1] **mod** 16)
define *italic_index*(#) ≡ (*tfm*[*char_info*(#) + 2] **div** 4)
define *tag*(#) ≡ (*tfm*[*char_info*(#) + 2] **mod** 4)
define *reset_tag*(#) ≡ *tfm*[*char_info*(#) + 2] ← 4 ∗ *italic_index*(#) + *no_tag*
define *remainder*(#) ≡ *tfm*[*char_info*(#) + 3]
define *width*(#) ≡ 4 ∗ (*width_base* + *width_index*(#))
define *height*(#) ≡ 4 ∗ (*height_base* + *height_index*(#))
define *depth*(#) ≡ 4 ∗ (*depth_base* + *depth_index*(#))
define *italic*(#) ≡ 4 ∗ (*italic_base* + *italic_index*(#))
define *exten*(#) ≡ 4 ∗ (*exten_base* + *remainder*(#))
define *lig_step*(#) ≡ 4 ∗ (*lig_kern_base* + (#))
define *kern*(#) ≡ 4 ∗ (*kern_base* + #) { here # is an index, not a character }
define *param*(#) ≡ 4 ∗ (*param_base* + #) { likewise }

25. One of the things we would like to do is take cognizance of fonts whose character coding scheme is `TeX math symbols` or `TeX math extension`; we will set the *font_type* variable to one of the three choices *vanilla*, *mathsy*, or *mathex*.

define *vanilla* = 0 { not a special scheme }
define *mathsy* = 1 { `TeX math symbols` scheme }
define *mathex* = 2 { `TeX math extension` scheme }

⟨ Globals in the outer block 6 ⟩ +≡
font_type: *vanilla* .. *mathex*; { is this font special? }

26. Basic output subroutines. Let us now define some procedures that will reduce the rest of TFtoPL's work to a triviality.

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

define *out*(#) ≡ *write*(*pl_file*, #)

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*: **packed array** [1 .. 32] **of** *char*;
{ strings for output in the user's external character set }
MBL_string, *RI_string*, *RCE_string*: **packed array** [1 .. 3] **of** *char*;
{ handy string constants for *face* codes }

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

29. The array *dig* will hold a sequence of digits to be output.

⟨ Globals in the outer block 6 ⟩ +≡
dig: **array** [0 .. 11] **of** 0 .. 9;

30. Here, in fact, are two procedures that output *dig*[*j* − 1] . . . *dig*[0], given *j* > 0.

procedure *out_digs*(*j* : *integer*); { outputs *j* digits }
begin repeat *decr*(*j*); *out*(*dig*[*j*] : 1);
until *j* = 0;
end;

procedure *print_digs*(*j* : *integer*); { prints *j* digits }
begin repeat *decr*(*j*); *print*(*dig*[*j*] : 1);
until *j* = 0;
end;

31. The *print_octal* procedure indicates how *print_digs* can be used. Since this procedure is used only to print character codes, it always produces three digits.

procedure *print_octal*(*c* : *byte*); { prints octal value of *c* }
var *j*: 0 .. 2; { index into *dig* }
begin *print*(´´´´); { an apostrophe indicates the octal notation }
for *j* ← 0 **to** 2 **do**
begin *dig*[*j*] ← *c* **mod** 8; *c* ← *c* **div** 8;
end;
print_digs(3);
end;

32. A PL file has nested parentheses, and we want to format the output so that its structure is clear. The *level* variable keeps track of the depth of nesting.

⟨ Globals in the outer block 6 ⟩ +≡
level: 0 .. 5;

33. ⟨ Set initial values 7 ⟩ +≡
```
level ← 0;
```

34. Three simple procedures suffice to produce the desired structure in the output.
```
procedure out_ln; { finishes one line, indents the next }
  var l: 0 .. 5;
  begin write_ln(pl_file);
  for l ← 1 to level do out('␣␣␣');
  end;
procedure left; { outputs a left parenthesis }
  begin incr(level); out('(');
  end;
procedure right; { outputs a right parenthesis and finishes a line }
  begin decr(level); out(')'); out_ln;
  end;
```

35. The value associated with a property can be output in a variety of ways. For example, we might want to output a BCPL string that begins in $tfm[k]$:
```
procedure out_BCPL(k : index); { outputs a string, preceded by a blank space }
  var l: 0 .. 39; { the number of bytes remaining }
  begin out('␣'); l ← tfm[k];
  while l > 0 do
    begin incr(k); decr(l);
    case tfm[k] div '40 of
    1: out(ASCII_04[1 + (tfm[k] mod '40)]);
    2: out(ASCII_10[1 + (tfm[k] mod '40)]);
    3: out(ASCII_14[1 + (tfm[k] mod '40)]);
    end;
    end;
  end;
```

36. The property value might also be a sequence of l bytes, beginning in $tfm[k]$, that we would like to output in octal notation. The following procedure assumes that $l \le 4$, but larger values of l could be handled easily by enlarging the *dig* array and increasing the upper bounds on b and j.
```
procedure out_octal(k, l : index); { outputs l bytes in octal }
  var a: 0 .. '1777; { accumulator for bits not yet output }
    b: 0 .. 32; { the number of significant bits in a }
    j: 0 .. 11; { the number of digits of output }
  begin out('␣0␣'); { specify octal format }
  a ← 0; b ← 0; j ← 0;
  while l > 0 do ⟨ Reduce l by one, preserving the invariants 37 ⟩;
  while (a > 0) ∨ (j = 0) do
    begin dig[j] ← a mod 8; a ← a div 8; incr(j);
    end;
  out_digs(j);
  end;
```

37. ⟨ Reduce l by one, preserving the invariants 37 ⟩ ≡

```
begin decr(l);
if tfm[k+l] ≠ 0 then
  begin while b > 2 do
    begin dig[j] ← a mod 8; a ← a div 8; b ← b − 3; incr(j);
    end;
  case b of
  0: a ← tfm[k+l];
  1: a ← a + 2 * tfm[k+l];
  2: a ← a + 4 * tfm[k+l];
  end;
  end;
b ← b + 8;
end
```

This code is used in section 36.

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 then
    begin tfm[0] ← c; out_octal(0, 1)
    end
  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; out(MBL_string[1 + (b mod 3)]); out(RI_string[1 + s]);
    out(RCE_string[1 + (b div 3)]);
    end;
  end;
```

40. And finally, the value might be a *fix_word*, which is output in decimal notation with just enough decimal places for `PLtoTF` to recover every bit of the given *fix_word*.

All of the numbers involved in the intermediate calculations of this procedure will be nonnegative and less than $10 \cdot 2^{24}$.

```
procedure out_fix(k : index);  { outputs a fix_word }
  var a: 0 .. ´7777;  { accumulator for the integer part }
    f: integer;  { accumulator for the fraction part }
    j: 0 .. 12;  { index into dig }
    delta: integer;  { amount if allowable inaccuracy }
  begin out(´␣R␣´);  { specify real format }
  a ← (tfm[k] * 16) + (tfm[k + 1] div 16); f ← ((tfm[k + 1] mod 16) * ´400 + tfm[k + 2]) * ´400 + tfm[k + 3];
  if a > ´3777 then ⟨ Reduce negative to positive 43 ⟩;
  ⟨ Output the integer part, a, in decimal notation 41 ⟩;
  ⟨ Output the fraction part, f/2^20, in decimal notation 42 ⟩;
  end;
```

41. The following code outputs at least one digit even if $a = 0$.

```
⟨ Output the integer part, a, in decimal notation 41 ⟩ ≡
  begin j ← 0;
  repeat dig[j] ← a mod 10; a ← a div 10; incr(j);
  until a = 0;
  out_digs(j);
  end
```

This code is used in section 40.

42. And the following code outputs at least one digit to the right of the decimal point.

```
⟨ Output the fraction part, f/2^20, in decimal notation 42 ⟩ ≡
  begin out(´.´); f ← 10 * f + 5; delta ← 10;
  repeat if delta > ´4000000 then f ← f + ´2000000 − (delta div 2);
    out(f div ´4000000 : 1); f ← 10 * (f mod ´4000000); delta ← delta * 10;
  until f ≤ delta;
  end;
```

This code is used in section 40.

43.

```
⟨ Reduce negative to positive 43 ⟩ ≡
  begin out(´-´); a ← ´10000 − a;
  if f > 0 then
    begin f ← ´4000000 − f; decr(a);
    end;
  end
```

This code is used in section 40.

44. Doing it. TEX checks the information of a `TFM` file for validity as the file is being read in, so that no further checks will be needed when typesetting is going on. And when it finds something wrong, it just calls the file "bad," without identifying the nature of the problem, since `TFM` files are supposed to be good almost all of the time.

Of course, a bad file shows up every now and again, and that's where `TFtoPL` comes in. This program wants to catch at least as many errors as TEX does, and to give informative error messages besides. All of the errors are corrected, so that the `PL` output will be correct (unless, of course, the `TFM` file was so loused up that no attempt is being made to fathom it).

45. Just before each character is processed, its 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. We also keep track of whether or not any errors have had to be corrected.

```
⟨ Globals in the outer block 6 ⟩ +≡
chars_on_line: 0 .. 8;  { the number of characters printed on the current line }
perfect: boolean;  { was the file free of errors? }
```

46.
```
⟨ Set initial values 7 ⟩ +≡
  chars_on_line ← 0;
  perfect ← true;  { innocent until proved guilty }
```

47. Error messages are given with the help of the *bad* and *range_error* and *bad_char* macros:

```
define bad(#) ≡
        begin perfect ← false;
        if chars_on_line > 0 then print_ln(´␣´);
        chars_on_line ← 0; print_ln(´Bad␣TFM␣file:␣´,#);
        end
define range_error(#) ≡
        begin perfect ← false; print_ln(´␣´); print(#,´␣index␣for␣character␣´); print_octal(c);
        print_ln(´␣is␣too␣large;´); print_ln(´so␣I␣reset␣it␣to␣zero.´);
        end
define bad_char_tail(#) ≡ print_octal(#); print_ln(´.´);
      end
define bad_char(#) ≡
      begin perfect ← false;
      if chars_on_line > 0 then print_ln(´␣´);
      chars_on_line ← 0; print(´Bad␣TFM␣file:␣´,#,´␣nonexistent␣character␣´); bad_char_tail
define correct_bad_char_tail(#) ≡ print_octal(tfm[#]); print_ln(´.´); tfm[#] ← bc;
      end
define correct_bad_char(#) ≡
      begin perfect ← false;
      if chars_on_line > 0 then print_ln(´␣´);
      chars_on_line ← 0; print(´Bad␣TFM␣file:␣´,#,´␣nonexistent␣character␣´);
      correct_bad_char_tail
```

```
⟨ Globals in the outer block 6 ⟩ +≡
i: 0 .. ´77777;  { an index to words of a subfile }
c: 0 .. 256;  { a random character }
d: 0 .. 3;  { byte number in a word }
k: index;  { a random index }
r: 0 .. 65535;  { a random two-byte value }
count: 0 .. 127;  { for when we need to enumerate a small set }
```

48. There are a lot of simple things to do, and they have to be done one at a time, so we might as well get down to business. The first things that TFtoPL will put into the PL file appear in the header part.

```
⟨ Do the header 48 ⟩ ≡
  begin font_type ← vanilla;
  if lh ≥ 12 then
    begin ⟨ Set the true font_type 53 ⟩;
    if lh ≥ 17 then
      begin ⟨ Output the family name 55 ⟩;
      if lh ≥ 18 then ⟨ Output the rest of the header 56 ⟩;
      end;
    ⟨ Output the character coding scheme 54 ⟩;
    end;
  ⟨ Output the design size 51 ⟩;
  ⟨ Output the check sum 49 ⟩;
  ⟨ Output the seven_bit_safe_flag 57 ⟩;
  end
```

This code is used in section 97.

49.
```
⟨ Output the check sum 49 ⟩ ≡
  left; out(´CHECKSUM´); out_octal(check_sum, 4); right
```

This code is used in section 48.

50. Incorrect design sizes are changed to 10 points.

```
define bad_design(#) ≡
         begin bad(´Design␣size␣´, #, ´!´); print_ln(´I´´ve␣set␣it␣to␣10␣points.´);
         out(´␣D␣10´);
         end
```

51.
```
⟨ Output the design size 51 ⟩ ≡
  left; out(´DESIGNSIZE´);
  if tfm[design_size] > 127 then bad_design(´negative´)
  else if (tfm[design_size] = 0) ∧ (tfm[design_size + 1] < 16) then bad_design(´too␣small´)
    else out_fix(design_size);
  right; out(´(COMMENT␣DESIGNSIZE␣IS␣IN␣POINTS)´); out_ln;
  out(´(COMMENT␣OTHER␣SIZES␣ARE␣MULTIPLES␣OF␣DESIGNSIZE)´); out_ln
```

This code is used in section 48.

52. Since we have to check two different BCPL strings for validity, we might as well write a subroutine to make the check.

```
procedure check_BCPL(k, l : index);  { checks a string of length < l }
  var j: index;  { runs through the string }
    c: byte;  { character being checked }
  begin if tfm[k] ≥ l then
    begin bad('String␣is␣too␣long;␣I''ve␣shortened␣it␣drastically.'); tfm[k] ← 1;
    end;
  for j ← k + 1 to k + tfm[k] do
    begin c ← tfm[j];
    if (c = "(") ∨ (c = ")") then
      begin bad('Parenthesis␣in␣string␣has␣been␣changed␣to␣slash.'); tfm[j] ← "/";
      end
    else if (c < "␣") ∨ (c > "~") then
        begin bad('Nonstandard␣ASCII␣code␣has␣been␣blotted␣out.'); tfm[j] ← "?";
        end
      else if (c ≥ "a") ∧ (c ≤ "z") then tfm[j] ← c + "A" − "a";  { upper-casify letters }
    end;
  end;
```

53. The *font_type* starts out *vanilla*; possibly we need to reset it.

```
⟨ Set the true font_type 53 ⟩ ≡
  begin check_BCPL(scheme, 40);
  if (tfm[scheme] ≥ 11) ∧ (tfm[scheme + 1] = "T") ∧ (tfm[scheme + 2] = "E") ∧ (tfm[scheme + 3] = "X") ∧
        (tfm[scheme + 4] = "␣") ∧ (tfm[scheme + 5] = "M") ∧ (tfm[scheme + 6] = "A") ∧
        (tfm[scheme + 7] = "T") ∧ (tfm[scheme + 8] = "H") ∧ (tfm[scheme + 9] = "␣") then
    begin if (tfm[scheme + 10] = "S") ∧ (tfm[scheme + 11] = "Y") then font_type ← mathsy
    else if (tfm[scheme + 10] = "E") ∧ (tfm[scheme + 11] = "X") then font_type ← mathex;
    end;
  end
```

This code is used in section 48.

54. ⟨ Output the character coding scheme 54 ⟩ ≡

```
left; out('CODINGSCHEME'); out_BCPL(scheme); right
```

This code is used in section 48.

55. ⟨ Output the family name 55 ⟩ ≡

```
left; out('FAMILY'); check_BCPL(family, 20); out_BCPL(family); right
```

This code is used in section 48.

56. ⟨ Output the rest of the header 56 ⟩ ≡

```
begin left; out('FACE'); out_face(random_word + 3); right;
for i ← 18 to lh − 1 do
  begin left; out('HEADER␣D␣', i : 1); out_octal(check_sum + 4 ∗ i, 4); right;
  end;
end
```

This code is used in section 48.

57. This program does not check to see if the *seven_bit_safe_flag* has the correct setting, i.e., if it really reflects the seven-bit-safety of the TFM file; the stated value is merely put into the PL file. The PLtoTF program will store a correct value and give a warning message if a file falsely claims to be safe.

```
⟨ Output the seven_bit_safe_flag 57 ⟩ ≡
  if (lh > 17) ∧ (tfm[random_word] > 127) then
    begin left; out('SEVENBITSAFEFLAG␣TRUE'); right;
    end
```

This code is used in section 48.

58. The next thing to take care of is the list of parameters.

```
⟨ Do the parameters 58 ⟩ ≡
  if np > 0 then
    begin left; out('FONTDIMEN'); out_ln;
    for i ← 1 to np do ⟨ Check and output the ith parameter 60 ⟩;
    right;
    end;
  ⟨ Check to see if np is complete for this font type 59 ⟩;
```

This code is used in section 97.

59.

```
⟨ Check to see if np is complete for this font type 59 ⟩ ≡
  if (font_type = mathsy) ∧ (np ≠ 22) then
    print_ln('Unusual␣number␣of␣fontdimen␣parameters␣for␣a␣math␣symbols␣font␣(', np : 1,
        '␣not␣22).')
  else if (font_type = mathex) ∧ (np ≠ 13) then
      print_ln('Unusual␣number␣of␣fontdimen␣parameters␣for␣an␣extension␣font␣(', np : 1,
          '␣not␣13).')
```

This code is used in section 58.

60. All *fix_word* values except the design size and the first parameter will be checked to make sure that they are less than 16.0 in magnitude, using the *check_fix* macro:

```
define check_fix_tail(#) ≡ bad(#, '␣', i : 1, '␣is␣too␣big;'); print_ln('I␣have␣set␣it␣to␣zero.');
        end
define check_fix(#) ≡
        if (tfm[#] > 0) ∧ (tfm[#] < 255) then
          begin tfm[#] ← 0; tfm[(#) + 1] ← 0; tfm[(#) + 2] ← 0; tfm[(#) + 3] ← 0; check_fix_tail
```

```
⟨ Check and output the ith parameter 60 ⟩ ≡
  begin left;
  if i = 1 then out('SLANT')   { this parameter is not checked }
  else begin check_fix(param(i))('Parameter');
    ⟨ Output the name of parameter i 61 ⟩;
    end;
  out_fix(param(i)); right;
  end
```

This code is used in section 58.

61. ⟨ Output the name of parameter i 61 ⟩ ≡

```
if i ≤ 7 then
  case i of
  2: out(´SPACE´); 3: out(´STRETCH´); 4: out(´SHRINK´);
  5: out(´XHEIGHT´); 6: out(´QUAD´); 7: out(´EXTRASPACE´)
  end
else if (i ≤ 22) ∧ (font_type = mathsy) then
    case i of
    8: out(´NUM1´); 9: out(´NUM2´); 10: out(´NUM3´);
    11: out(´DENOM1´); 12: out(´DENOM2´);
    13: out(´SUP1´); 14: out(´SUP2´); 15: out(´SUP3´);
    16: out(´SUB1´); 17: out(´SUB2´);
    18: out(´SUPDROP´); 19: out(´SUBDROP´);
    20: out(´DELIM1´); 21: out(´DELIM2´);
    22: out(´AXISHEIGHT´)
    end
  else if (i ≤ 13) ∧ (font_type = mathex) then
      if i = 8 then out(´DEFAULTRULETHICKNESS´)
      else out(´BIGOPSPACING´, i − 8 : 1)
    else out(´PARAMETER␣D␣´, i : 1)
```

This code is used in section 60.

62. We need to check the range of all the remaining *fix_word* values, and to make sure that $width[0] = 0$, etc.

```
define nonzero_fix(#) ≡ (tfm[#] > 0) ∨ (tfm[# + 1] > 0) ∨ (tfm[# + 2] > 0) ∨ (tfm[# + 3] > 0)
```

⟨ Check the *fix_word* entries 62 ⟩ ≡

```
if nonzero_fix(4 ∗ width_base) then bad(´width[0]␣should␣be␣zero.´);
if nonzero_fix(4 ∗ height_base) then bad(´height[0]␣should␣be␣zero.´);
if nonzero_fix(4 ∗ depth_base) then bad(´depth[0]␣should␣be␣zero.´);
if nonzero_fix(4 ∗ italic_base) then bad(´italic[0]␣should␣be␣zero.´);
for i ← 0 to nw − 1 do check_fix(4 ∗ (width_base + i))(´Width´);
for i ← 0 to nh − 1 do check_fix(4 ∗ (height_base + i))(´Height´);
for i ← 0 to nd − 1 do check_fix(4 ∗ (depth_base + i))(´Depth´);
for i ← 0 to ni − 1 do check_fix(4 ∗ (italic_base + i))(´Italic␣correction´);
if nk > 0 then
  for i ← 0 to nk − 1 do check_fix(kern(i))(´Kern´);
```

This code is used in section 97.

63. The ligature/kerning program comes next. Before we can put it out in PL format, we need to make a table of "labels" that will be inserted into the program. For each character c whose *tag* is *lig_tag* and whose starting address is r, we will store the pair (c, r) in the *label_table* array. If there's a boundary-char program starting at r, we also store the pair $(256, r)$. This array is sorted by its second components, using the simple method of straight insertion.

```
⟨ Globals in the outer block 6 ⟩ +≡
label_table: array [0 .. 258] of record
    cc: 0 .. 256;
    rr: 0 .. lig_size;
    end;
label_ptr: 0 .. 257;  { the largest entry in label_table }
sort_ptr: 0 .. 257;  { index into label_table }
boundary_char: 0 .. 256;  { boundary character, or 256 if none }
bchar_label: 0 .. '77777;  { beginning of boundary character program }
```

64.
```
⟨ Set initial values 7 ⟩ +≡
  boundary_char ← 256; bchar_label ← '77777;
  label_ptr ← 0; label_table[0].rr ← 0;  { a sentinel appears at the bottom }
```

65. We'll also identify and remove inaccessible program steps, using the *activity* array.

```
  define unreachable = 0  { a program step not known to be reachable }
  define pass_through = 1  { a program step passed through on initialization }
  define accessible = 2  { a program step that can be relevant }
⟨ Globals in the outer block 6 ⟩ +≡
activity: array [0 .. lig_size] of unreachable .. accessible;
ai, acti: 0 .. lig_size;  { indices into activity }
```

66.
```
⟨ Do the ligatures and kerns 66 ⟩ ≡
  if nl > 0 then
    begin for ai ← 0 to nl − 1 do activity[ai] ← unreachable;
    ⟨ Check for a boundary char 69 ⟩;
    end;
  ⟨ Build the label table 67 ⟩;
  if nl > 0 then
    begin left; out('LIGTABLE'); out_ln;
    ⟨ Compute the activity array 70 ⟩;
    ⟨ Output and correct the ligature/kern program 71 ⟩;
    right; ⟨ Check for ligature cycles 90 ⟩;
    end
```

This code is used in section 99.

67. We build the label table even when $nl = 0$, because this catches errors that would not otherwise be detected.

```
⟨ Build the label table 67 ⟩ ≡
  for c ← bc to ec do
    if tag(c) = lig_tag then
      begin r ← remainder(c);
      if r < nl then
        begin if tfm[lig_step(r)] > stop_flag then
          begin r ← 256 * tfm[lig_step(r) + 2] + tfm[lig_step(r) + 3];
          if r < nl then
            if activity[remainder(c)] = unreachable then activity[remainder(c)] ← pass_through;
          end;
        end;
      if r ≥ nl then
        begin perfect ← false; print_ln('␣'); print('Ligature/kern␣starting␣index␣for␣character␣');
        print_octal(c); print_ln('␣is␣too␣large;'); print_ln('so␣I␣removed␣it.'); reset_tag(c);
        end
      else ⟨ Insert (c, r) into label_table 68 ⟩;
      end;
  label_table[label_ptr + 1].rr ← lig_size; { put "infinite" sentinel at the end }
```

This code is used in section 66.

68. ⟨ Insert (c, r) into *label_table* 68 ⟩ ≡

```
  begin sort_ptr ← label_ptr; { there's a hole at position sort_ptr + 1 }
  while label_table[sort_ptr].rr > r do
    begin label_table[sort_ptr + 1] ← label_table[sort_ptr]; decr(sort_ptr); { move the hole }
    end;
  label_table[sort_ptr + 1].cc ← c; label_table[sort_ptr + 1].rr ← r; { fill the hole }
  incr(label_ptr); activity[r] ← accessible;
  end
```

This code is used in section 67.

69. ⟨ Check for a boundary char 69 ⟩ ≡

```
  if tfm[lig_step(0)] = 255 then
    begin left; out('BOUNDARYCHAR'); boundary_char ← tfm[lig_step(0) + 1]; out_char(boundary_char);
    right; activity[0] ← pass_through;
    end;
  if tfm[lig_step(nl − 1)] = 255 then
    begin r ← 256 * tfm[lig_step(nl − 1) + 2] + tfm[lig_step(nl − 1) + 3];
    if r ≥ nl then
      begin perfect ← false; print_ln('␣'); print('Ligature/kern␣starting␣index␣for␣boundarychar␣is␣too␣large;');
      print_ln('so␣I␣removed␣it.');
      end
    else begin label_ptr ← 1; label_table[1].cc ← 256; label_table[1].rr ← r; bchar_label ← r;
      activity[r] ← accessible;
      end;
    activity[nl − 1] ← pass_through;
    end
```

This code is used in section 66.

70. ⟨ Compute the *activity* array 70 ⟩ ≡

```
for ai ← 0 to nl − 1 do
  if activity[ai] = accessible then
    begin r ← tfm[lig_step(ai)];
    if r < stop_flag then
      begin r ← r + ai + 1;
      if r ≥ nl then
        begin bad(´Ligature/kern␣step␣´, ai : 1, ´␣skips␣too␣far;´);
        print_ln(´I␣made␣it␣stop.´); tfm[lig_step(ai)] ← stop_flag;
        end
      else activity[r] ← accessible;
      end;
    end
```

This code is used in section 66.

71. We ignore *pass_through* items, which don't need to be mentioned in the PL file.

⟨ Output and correct the ligature/kern program 71 ⟩ ≡

```
sort_ptr ← 1;  { point to the next label that will be needed }
for acti ← 0 to nl − 1 do
  if activity[acti] ≠ pass_through then
    begin i ← acti; ⟨ Take care of commenting out unreachable steps 73 ⟩;
    ⟨ Output any labels for step i 72 ⟩;
    ⟨ Output step i of the ligature/kern program 74 ⟩;
    end;
if level = 2 then right  { the final step was unreachable }
```

This code is used in section 66.

72. ⟨ Output any labels for step *i* 72 ⟩ ≡

```
while i = label_table[sort_ptr].rr do
  begin left; out(´LABEL´);
  if label_table[sort_ptr].cc = 256 then out(´␣BOUNDARYCHAR´)
  else out_char(label_table[sort_ptr].cc);
  right; incr(sort_ptr);
  end
```

This code is used in section 71.

73. ⟨ Take care of commenting out unreachable steps 73 ⟩ ≡

```
if activity[i] = unreachable then
  begin if level = 1 then
    begin left; out(´COMMENT␣THIS␣PART␣OF␣THE␣PROGRAM␣IS␣NEVER␣USED!´); out_ln;
    end
  end
else if level = 2 then right
```

This code is used in section 71.

74. ⟨ Output step i of the ligature/kern program 74 ⟩ ≡

```
begin k ← lig_step(i);
if tfm[k] > stop_flag then
  begin if 256 * tfm[k + 2] + tfm[k + 3] ≥ nl then
    bad(´Ligature␣unconditional␣stop␣command␣address␣is␣too␣big.´);
  end
else if tfm[k + 2] ≥ kern_flag then ⟨ Output a kern step 76 ⟩
  else ⟨ Output a ligature step 77 ⟩;
if tfm[k] > 0 then
  if level = 1 then ⟨ Output either SKIP or STOP 75 ⟩;
end
```

This code is used in sections 71 and 83.

75. The SKIP command is a bit tricky, because we will be omitting all inaccessible commands.

⟨ Output either SKIP or STOP 75 ⟩ ≡

```
begin if tfm[k] ≥ stop_flag then out(´(STOP)´)
else begin count ← 0;
  for ai ← i + 1 to i + tfm[k] do
    if activity[ai] = accessible then incr(count);
  out(´(SKIP␣D␣´, count : 1, ´)´); { possibly count = 0, so who cares }
  end;
out_ln;
end
```

This code is used in section 74.

76. ⟨ Output a kern step 76 ⟩ ≡

```
begin if nonexistent(tfm[k + 1]) then
  if tfm[k + 1] ≠ boundary_char then correct_bad_char(´Kern␣step␣for´)(k + 1);
left; out(´KRN´); out_char(tfm[k + 1]); r ← 256 * (tfm[k + 2] − kern_flag) + tfm[k + 3];
if r ≥ nk then
  begin bad(´Kern␣index␣too␣large.´); out(´␣R␣0.0´);
  end
else out_fix(kern(r));
right;
end
```

This code is used in section 74.

77. ⟨ Output a ligature step 77 ⟩ ≡

```
begin if nonexistent(tfm[k+1]) then
  if tfm[k+1] ≠ boundary_char then correct_bad_char('Ligature␣step␣for')(k+1);
if nonexistent(tfm[k+3]) then correct_bad_char('Ligature␣step␣produces␣the')(k+3);
left; r ← tfm[k+2];
if (r = 4) ∨ ((r > 7) ∧ (r ≠ 11)) then
  begin print_ln('Ligature␣step␣with␣nonstandard␣code␣changed␣to␣LIG'); r ← 0; tfm[k+2] ← 0;
  end;
if r mod 4 > 1 then out('/');
out('LIG');
if odd(r) then out('/');
while r > 3 do
  begin out('>'); r ← r − 4;
  end;
out_char(tfm[k+1]); out_char(tfm[k+3]); right;
end
```

This code is used in section 74.

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('␣');
      incr(chars_on_line);
      end;
    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.

79. ⟨ Output the character's width 79 ⟩ ≡

```
begin left; out('CHARWD');
if width_index(c) ≥ nw then range_error('Width')
else out_fix(width(c));
right;
end
```

This code is used in section 78.

80. ⟨ Output the character's height 80 ⟩ ≡
```
if height_index(c) ≥ nh then range_error('Height')
else begin left; out('CHARHT'); out_fix(height(c)); right;
  end
```
This code is used in section 78.

81. ⟨ Output the character's depth 81 ⟩ ≡
```
if depth_index(c) ≥ nd then range_error('Depth')
else begin left; out('CHARDP'); out_fix(depth(c)); right;
  end
```
This code is used in section 78.

82. ⟨ Output the italic correction 82 ⟩ ≡
```
if italic_index(c) ≥ ni then range_error('Italic␣correction')
else begin left; out('CHARIC'); out_fix(italic(c)); right;
  end
```
This code is used in section 78.

83. ⟨ Output the applicable part of the ligature/kern program as a comment 83 ⟩ ≡
```
begin left; out('COMMENT'); out_ln;
i ← remainder(c); r ← lig_step(i);
if tfm[r] > stop_flag then i ← 256 * tfm[r + 2] + tfm[r + 3];
repeat ⟨ Output step i of the ligature/kern program 74 ⟩;
  if tfm[k] ≥ stop_flag then i ← nl
  else i ← i + 1 + tfm[k];
until i ≥ nl;
right;
end
```
This code is used in section 78.

84. We want to make sure that there is no cycle of characters linked together by *list_tag* entries, since TEX doesn't want to risk endless loops. If such a cycle exists, the routine here detects it when processing the largest character code in the cycle.

⟨ Output the character link unless there is a problem 84 ⟩ ≡
```
begin r ← remainder(c);
if nonexistent(r) then
  begin bad_char('Character␣list␣link␣to')(r); reset_tag(c);
  end
else begin while (r < c) ∧ (tag(r) = list_tag) do r ← remainder(r);
  if r = c then
    begin bad('Cycle␣in␣a␣character␣list!'); print('Character␣'); print_octal(c);
    print_ln('␣now␣ends␣the␣list.'); reset_tag(c);
    end
  else begin left; out('NEXTLARGER'); out_char(remainder(c)); right;
    end;
  end;
end
```
This code is used in section 78.

85. ⟨ Output an extensible character recipe 85 ⟩ ≡

```
if remainder(c) ≥ ne then
  begin range_error(´Extensible´); reset_tag(c);
  end
else begin left; out(´VARCHAR´); out_ln; ⟨ Output the extensible pieces that exist 86 ⟩;
  right;
  end
```

This code is used in section 78.

86. ⟨ Output the extensible pieces that exist 86 ⟩ ≡

```
for k ← 0 to 3 do
  if (k = 3) ∨ (tfm[exten(c) + k] > 0) then
    begin left;
    case k of
    0: out(´TOP´); 1: out(´MID´); 2: out(´BOT´); 3: out(´REP´)
    end;
    if nonexistent(tfm[exten(c) + k]) then out_char(c)
    else out_char(tfm[exten(c) + k]);
    right;
    end
```

This code is used in section 85.

87. Some of the extensible recipes may not actually be used, but TEX will complain about them anyway if they refer to nonexistent characters. Therefore TFtoPL must check them too.

⟨ Check the extensible recipes 87 ⟩ ≡

```
if ne > 0 then
  for c ← 0 to ne − 1 do
    for d ← 0 to 3 do
      begin k ← 4 ∗ (exten_base + c) + d;
      if (tfm[k] > 0) ∨ (d = 3) then
        begin if nonexistent(tfm[k]) then
          begin bad_char(´Extensible␣recipe␣involves␣the´)(tfm[k]);
          if d < 3 then tfm[k] ← 0;
          end;
        end;
      end
```

This code is used in section 99.

88. Checking for ligature loops. We have programmed almost everything but the most interesting calculation of all, which has been saved for last as a special treat. TEX's extended ligature mechanism allows unwary users to specify sequences of ligature replacements that never terminate. For example, the pair of commands

`(/LIG` x y`)` `(/LIG` y x`)`

alternately replaces character x by character y and vice versa. A similar loop occurs if `(LIG/` z y`)` occurs in the program for x and `(LIG/` z x`)` occurs in the program for y.

More complicated loops are also possible. For example, suppose the ligature programs for x and y are

`(LABEL` x`)(/LIG/` z w`)(/LIG/>` w y`)` ...,
`(LABEL` y`)(LIG` w x`)` ...;

then the adjacent characters xz change to xwz, $xywz$, xxz, $xxwz$, ..., ad infinitum.

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 }

⟨ 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!'); goto final_end;
  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(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!)'); goto final_end;
  end
```

This code is used in section 66.

91. ⟨ Enter data for character c starting at location i in the hash table 91 ⟩ ≡

```
repeat hash_input; k ← tfm[lig_step(i)];
  if k ≥ stop_flag then i ← nl
  else i ← i + 1 + k;
until i ≥ nl
```

This code is used in sections 90 and 90.

92. We use an "ordered hash table" with linear probing, because such a table is efficient when the lookup of a random key tends to be unsuccessful.

```
procedure hash_input; { enter data for character c and command i }
  label 30; { go here for a quick exit }
  var cc: simple .. both_z; { class of data being entered }
    zz: 0 .. 255; { function value or ligature character being entered }
    y: 0 .. 255; { the character after the cursor }
    key: integer; { value to be stored in hash }
    t: integer; { temporary register for swapping }
  begin if hash_ptr = hash_size then goto 30;
  ⟨ Compute the command parameters y, cc, and zz 93 ⟩;
  key ← 256 * c + y + 1; h ← (1009 * key) mod hash_size;
  while hash[h] > 0 do
    begin if hash[h] ≤ key then
      begin if hash[h] = key then goto 30; { unused ligature command }
      t ← hash[h]; hash[h] ← key; key ← t; { do ordered-hash-table insertion }
      t ← class[h]; class[h] ← cc; cc ← t; { namely, do a swap }
      t ← lig_z[h]; lig_z[h] ← zz; zz ← t;
      end;
    if h > 0 then decr(h) else h ← hash_size;
    end;
  hash[h] ← key; class[h] ← cc; lig_z[h] ← zz; incr(hash_ptr); hash_list[hash_ptr] ← h;
30: end;
```

93. We must store kern commands as well as ligature commands, because the former might make the latter inapplicable.

```
⟨ Compute the command parameters y, cc, and zz 93 ⟩ ≡
  k ← lig_step(i); y ← tfm[k + 1]; t ← tfm[k + 2]; cc ← simple; zz ← tfm[k + 3];
  if t ≥ kern_flag then zz ← y
  else begin case t of
    0, 6: do_nothing; { LIG,/LIG> }
    5, 11: zz ← y; { LIG/>, /LIG/>> }
    1, 7: cc ← left_z; { LIG/, /LIG/> }
    2: cc ← right_z; { /LIG }
    3: cc ← both_z; { /LIG/ }
    end; { there are no other cases }
    end
```

This code is used in section 92.

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

```
function f(h, x, y : index): index; forward; { compute f for arguments known to be in hash[h] }
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(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;
  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;
```

96. The main program. The routines sketched out so far need to be packaged into separate procedures, on some systems, since some Pascal compilers place a strict limit on the size of a routine. The packaging is done here in an attempt to avoid some system-dependent changes.

First comes the *organize* procedure, which reads the input data and gets ready for subsequent events. If something goes wrong, the routine returns *false*.

```
function organize: boolean;
  label final_end, 30;
  var tfm_ptr: index;  { an index into tfm }
  begin ⟨ Read the whole input file 20 ⟩;
  ⟨ Set subfile sizes lh, bc, ..., np 21 ⟩;
  ⟨ Compute the base addresses 23 ⟩;
  organize ← true; goto 30;
final_end: organize ← false;
30: end;
```

97. Next we do the simple things.

```
procedure do_simple_things;
  var i: 0 .. '77777;  { an index to words of a subfile }
  begin ⟨ Do the header 48 ⟩;
  ⟨ Do the parameters 58 ⟩;
  ⟨ Check the fix_word entries 62 ⟩
  end;
```

98. And then there's a routine for individual characters.

```
procedure do_characters;
  var c: byte;  { character being done }
    k: index;  { a random index }
    ai: 0 .. lig_size;  { index into activity }
  begin ⟨ Do the characters 78 ⟩;
  end;
```

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

```
  begin initialize;
  if ¬organize then goto final_end;
  do_simple_things;
  ⟨ Do the ligatures and kerns 66 ⟩;
  ⟨ Check the extensible recipes 87 ⟩;
  do_characters; 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;
final_end: end.
```

100. System-dependent changes. This section should be replaced, if necessary, by changes to the program that are necessary to make `TFtoPL` work at a particular installation. It is usually best to design your change file so that all changes to previous sections preserve the section numbering; then everybody's version will be consistent with the printed program. More extensive changes, which introduce new sections, can be inserted here; then only the index itself will get a new section number.

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

a: 36, 40.
abort: 20, 21.
accessible: 65, 68, 69, 70, 75.
acti: 65, 71.
activity: 65, 66, 67, 68, 69, 70, 71, 73, 75, 98.
ai: 65, 66, 70, 75, 98.
ASCII_04: 27, 28, 35.
ASCII_10: 27, 28, 35, 38.
ASCII_14: 27, 28, 35, 38.
axis_height: 15.
b: 36, 39.
bad: 47, 50, 52, 60, 62, 70, 74, 76, 84.
`Bad TFM file`: 47.
bad_char: 47, 84, 87.
bad_char_tail: 47.
bad_design: 50, 51.
banner: 1, 2.
bc: 8, 9, 11, 13, 21, 23, 24, 47, 67, 78, 90.
bchar_label: 63, 64, 69, 90.
big_op_spacing1: 15.
big_op_spacing5: 15.
boolean: 45, 96.
bot: 14.
both_z: 89, 92, 93, 95.
boundary_char: 63, 64, 69, 76, 77.
byte: 18, 19, 31, 38, 52, 98.
c: 38, 47, 52, 98.
cc: 63, 68, 69, 72, 92, 93.
char: 27.
char_base: 22, 23, 24.
char_info: 11, 22, 24, 78.
char_info_word: 9, 11, 12.
`Character list link...`: 84.
chars_on_line: 45, 46, 47, 78.
check sum: 10.
check_BCPL: 52, 53, 55.
check_fix: 60, 62.
check_fix_tail: 60.
check_sum: 24, 49, 56.
class: 89, 90, 92, 95.
coding scheme: 10.
correct_bad_char: 47, 76, 77.
correct_bad_char_tail: 47.
count: 47, 75.
`Cycle in a character list`: 84.
d: 47.
decr: 5, 30, 34, 35, 37, 43, 68, 92, 94.
default_rule_thickness: 15.
delim1: 15.
delim2: 15.
delta: 40, 42.
denom1: 15.
denom2: 15.
depth: 11, 24, 81.
`Depth index for char`: 81.
`Depth n is too big`: 62.
depth_base: 22, 23, 24, 62.
depth_index: 11, 24, 78, 81.
design size: 10.
`Design size wrong`: 50.
design_size: 24, 51.
`DESIGNSIZE IS IN POINTS`: 51.
dig: 29, 30, 31, 36, 37, 40, 41.
do_characters: 98, 99.
do_nothing: 5, 78, 93, 95.
do_simple_things: 97, 99.
ec: 8, 9, 11, 13, 21, 23, 24, 67, 78, 90.
eof: 20.
eval: 94, 95.
eval_two_bytes: 21.
ext_tag: 12, 78.
exten: 12, 24, 86.
exten_base: 22, 23, 24, 87.
`Extensible index for char`: 85.
`Extensible recipe involves...`: 87.
extensible_recipe: 9, 14.
extra_space: 15.
f: 40, 94, 95.
face: 10, 27, 39.
false: 47, 67, 69, 96.
family: 24, 55.
family name: 10.
final_end: 3, 20, 90, 96, 99.
fix_word: 9, 10, 15, 24, 40, 60, 62.
font identifier: 10.
font_type: 25, 38, 48, 53, 59, 61.
forward: 94, 95.
h: 89, 94.
hash: 89, 90, 92, 94, 95.
hash_input: 91, 92.
hash_list: 89, 90, 92.
hash_ptr: 89, 90, 92.
hash_size: 4, 89, 90, 92, 94.
header: 10.
height: 11, 24, 80.
`Height index for char...`: 80.
`Height n is too big`: 62.
height_base: 22, 23, 24, 62.
height_index: 11, 24, 78, 80.
hh: 89, 90.

i: 47, 97.
Incomplete subfiles...: 21.
incr: 5, 34, 35, 36, 37, 41, 68, 72, 75, 78, 92.
index: 18, 35, 36, 39, 40, 47, 52, 94, 95, 96, 98.
Infinite ligature loop...: 90.
initialize: 2, 99.
integer: 22, 30, 40, 92, 94.
italic: 11, 24, 82.
Italic correction index for char...: 82.
Italic correction n is too big: 62.
italic_base: 22, 23, 24, 62.
italic_index: 11, 24, 78, 82.
j: 31, 36, 40, 52.
k: 35, 36, 39, 40, 47, 52, 98.
kern: 13, 24, 62, 76.
Kern index too large: 76.
Kern n is too big: 62.
Kern step for nonexistent...: 76.
kern_base: 22, 23, 24.
kern_flag: 13, 74, 76, 93.
key: 92, 94.
l: 34, 35, 36, 52.
label_ptr: 63, 64, 67, 68, 69.
label_table: 63, 64, 67, 68, 69, 72.
left: 34, 49, 51, 54, 55, 56, 57, 58, 60, 66, 69, 72, 73, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86.
left_z: 89, 93, 95.
level: 32, 33, 34, 71, 73, 74, 99.
lf: 8, 18, 20, 21.
lh: 8, 9, 21, 23, 48, 56, 57.
Lig...skips too far: 70.
lig_kern: 4, 12, 13.
lig_kern_base: 22, 23, 24.
lig_kern_command: 9, 13.
lig_size: 4, 21, 63, 65, 67, 98.
lig_step: 24, 67, 69, 70, 74, 83, 90, 91, 93.
lig_tag: 12, 63, 67, 78, 90.
lig_z: 89, 92, 95.
Ligature step for nonexistent...: 77.
Ligature step produces...: 77.
Ligature unconditional stop...: 74.
Ligature/kern starting index...: 67, 69.
list_tag: 12, 78, 84.
mathex: 25, 53, 59, 61.
mathsy: 25, 53, 59, 61.
MBL_string: 27, 28, 39.
mid: 14.
nd: 8, 9, 21, 23, 62, 81.
ne: 8, 9, 21, 23, 85, 87.
next_char: 13.
nh: 8, 9, 21, 23, 62, 80.
ni: 8, 9, 21, 23, 62, 82.
nk: 8, 9, 21, 23, 62, 76.
nl: 8, 9, 13, 21, 23, 66, 67, 69, 70, 71, 74, 83, 90, 91.
no_tag: 12, 24, 78.
nonexistent: 24, 76, 77, 84, 86, 87.
Nonstandard ASCII code...: 52.
nonzero_fix: 62.
np: 8, 9, 21, 58, 59.
num1: 15.
num2: 15.
num3: 15.
nw: 8, 9, 21, 23, 62, 79.
odd: 77.
One of the subfile sizes...: 21.
op_byte: 13.
organize: 96, 99.
out: 26, 30, 34, 35, 36, 38, 39, 40, 42, 43, 49, 50, 51, 54, 55, 56, 57, 58, 60, 61, 66, 69, 72, 73, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 90, 99.
out_BCPL: 35, 54, 55.
out_char: 38, 69, 72, 76, 77, 78, 84, 86.
out_digs: 30, 36, 41.
out_face: 39, 56.
out_fix: 40, 51, 60, 76, 79, 80, 81, 82.
out_ln: 34, 51, 58, 66, 73, 75, 78, 83, 85.
out_octal: 36, 38, 39, 49, 56.
output: 2.
param: 10, 15, 24, 60.
param_base: 22, 23, 24.
Parameter n is too big: 60.
Parenthesis...changed to slash: 52.
pass_through: 65, 67, 69, 71.
pending: 89, 95.
perfect: 45, 46, 47, 67, 69, 99.
pl_file: 2, 16, 17, 26, 34, 99.
print: 2, 30, 31, 47, 67, 69, 78, 84, 90.
print_digs: 30, 31.
print_ln: 2, 20, 47, 50, 59, 60, 67, 69, 70, 77, 78, 84, 90, 99.
print_octal: 31, 47, 67, 78, 84, 90.
quad: 15.
r: 47.
random_word: 24, 56, 57.
range_error: 47, 79, 80, 81, 82, 85.
RCE_string: 27, 28, 39.
read: 20.
remainder: 11, 12, 13, 24, 67, 83, 84, 85, 90.
rep: 14.
reset: 7.
reset_tag: 24, 67, 84, 85.
rewrite: 17.
RI_string: 27, 28, 39.

right: 34, 49, 51, 54, 55, 56, 57, 58, 60, 66, 69, 71, 72, 73, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86.
right_z: 89, 93, 95.
rr: 63, 64, 67, 68, 69, 72.
s: 39.
scheme: 24, 53, 54.
seven_bit_safe_flag: 10, 57.
`should be zero`: 62.
simple: 89, 90, 92, 93, 95.
skip_byte: 13.
slant: 15.
`Sorry, I haven't room...`: 90.
sort_ptr: 63, 68, 71, 72, 78.
space: 15.
space_shrink: 15.
space_stretch: 15.
stop_flag: 13, 67, 70, 74, 75, 83, 90, 91.
`String is too long...`: 52.
stuff: 9.
subdrop: 15.
`Subfile sizes don't add up...`: 21.
sub1: 15.
sub2: 15.
supdrop: 15.
sup1: 15.
sup2: 15.
sup3: 15.
system dependencies: 2, 7, 38, 100.
t: 92.
tag: 11, 12, 24, 63, 67, 78, 84, 90.
text: 16.
tfm: 4, 18, 19, 20, 21, 22, 24, 35, 36, 37, 38, 39, 40, 47, 51, 52, 53, 57, 60, 62, 67, 69, 70, 74, 75, 76, 77, 83, 86, 87, 90, 91, 93, 96.
tfm_file: 2, 6, 7, 18, 20.
tfm_ptr: 20, 21, 96.
tfm_size: 4, 18, 19, 20.
TFtoPL: 2.
`The character code range...`: 21.
`The file claims...`: 20.
`The file has fewer bytes...`: 20.
`The file is bigger...`: 20.
`The first byte...`: 20.
`The header length...`: 21.
`The input...one byte long`: 20.
`The lig/kern program...`: 21.
`THE TFM FILE WAS BAD...`: 99.
`There are ... recipes`: 21.
`There's some extra junk...`: 20.
`This program isn't working`: 99.
top: 14.
true: 46, 96.
unreachable: 65, 66, 67, 73.
`Unusual number of fontdimen...`: 59.
vanilla: 25, 38, 48, 53.
width: 11, 24, 62, 79.
`Width n is too big`: 62.
width_base: 22, 23, 24, 62.
width_index: 11, 24, 78, 79.
write: 2, 26.
write_ln: 2, 34, 99.
x: 94.
x_height: 15.
x_lig_cycle: 89, 90, 95.
y: 92, 94.
y_lig_cycle: 89, 90, 95.
zz: 92, 93.

⟨ Build the label table 67 ⟩ Used in section 66.
⟨ Check and output the ith parameter 60 ⟩ Used in section 58.
⟨ Check for a boundary char 69 ⟩ Used in section 66.
⟨ Check for ligature cycles 90 ⟩ Used in section 66.
⟨ Check the extensible recipes 87 ⟩ Used in section 99.
⟨ Check the *fix_word* entries 62 ⟩ Used in section 97.
⟨ Check to see if *np* is complete for this font type 59 ⟩ Used in section 58.
⟨ Compute the base addresses 23 ⟩ Used in section 96.
⟨ Compute the command parameters y, cc, and zz 93 ⟩ Used in section 92.
⟨ Compute the *activity* array 70 ⟩ Used in section 66.
⟨ Constants in the outer block 4 ⟩ Used in section 2.
⟨ Do the characters 78 ⟩ Used in section 98.
⟨ Do the header 48 ⟩ Used in section 97.
⟨ Do the ligatures and kerns 66 ⟩ Used in section 99.
⟨ Do the parameters 58 ⟩ Used in section 97.
⟨ Enter data for character c starting at location i in the hash table 91 ⟩ Used in sections 90 and 90.
⟨ Globals in the outer block 6, 8, 16, 19, 22, 25, 27, 29, 32, 45, 47, 63, 65, 89 ⟩ Used in section 2.
⟨ Insert (c, r) into *label_table* 68 ⟩ Used in section 67.
⟨ Labels in the outer block 3 ⟩ Used in section 2.
⟨ Output a kern step 76 ⟩ Used in section 74.
⟨ Output a ligature step 77 ⟩ Used in section 74.
⟨ Output an extensible character recipe 85 ⟩ Used in section 78.
⟨ Output and correct the ligature/kern program 71 ⟩ Used in section 66.
⟨ Output any labels for step i 72 ⟩ Used in section 71.
⟨ Output either `SKIP` or `STOP` 75 ⟩ Used in section 74.
⟨ Output step i of the ligature/kern program 74 ⟩ Used in sections 71 and 83.
⟨ Output the applicable part of the ligature/kern program as a comment 83 ⟩ Used in section 78.
⟨ Output the character coding scheme 54 ⟩ Used in section 48.
⟨ Output the character link unless there is a problem 84 ⟩ Used in section 78.
⟨ Output the character's depth 81 ⟩ Used in section 78.
⟨ Output the character's height 80 ⟩ Used in section 78.
⟨ Output the character's width 79 ⟩ Used in section 78.
⟨ Output the check sum 49 ⟩ Used in section 48.
⟨ Output the design size 51 ⟩ Used in section 48.
⟨ Output the extensible pieces that exist 86 ⟩ Used in section 85.
⟨ Output the family name 55 ⟩ Used in section 48.
⟨ Output the fraction part, $f/2^{20}$, in decimal notation 42 ⟩ Used in section 40.
⟨ Output the integer part, a, in decimal notation 41 ⟩ Used in section 40.
⟨ Output the italic correction 82 ⟩ Used in section 78.
⟨ Output the name of parameter i 61 ⟩ Used in section 60.
⟨ Output the rest of the header 56 ⟩ Used in section 48.
⟨ Output the *seven_bit_safe_flag* 57 ⟩ Used in section 48.
⟨ Read the whole input file 20 ⟩ Used in section 96.
⟨ Reduce l by one, preserving the invariants 37 ⟩ Used in section 36.
⟨ Reduce negative to positive 43 ⟩ Used in section 40.
⟨ Set initial values 7, 17, 28, 33, 46, 64 ⟩ Used in section 2.
⟨ Set subfile sizes lh, bc, ..., np 21 ⟩ Used in section 96.
⟨ Set the true *font_type* 53 ⟩ Used in section 48.
⟨ Take care of commenting out unreachable steps 73 ⟩ Used in section 71.
⟨ Types in the outer block 18 ⟩ Used in section 2.