April 1986

Report No. STAN-CS-86-1097

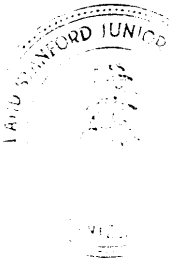
TEXware

by

Donald E. Knuth

Department of Computer Science

Stanford University
Stanford, CA 94305



The POOLtype processor

(Version 2, July 1983)

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 POOLtype utility program converts string pool files output by TANGLE into a slightly more symbolic format that may be useful when TANGLEd programs are being debugged.

It's a pretty trivial routine, but people may want to try transporting this program before they get up enough courage to tackle TEX itself. The first 128 strings are treated as TEX treats them, using routines copied from TEX82.

2. POOLtype is written entirely in standard Pascal, except that it has to do some slightly system-dependent character code conversion on input and output. The input is read from *pool_file*, and the output is written on *output*. If the input is erroneous, the *output* file will describe the error.

```
program POOLtype (pool-file, output);
  label 9999; { this labels the end of the program }
  type ⟨ Types in the outer block 5 ⟩
  var ⟨ Globals in the outer block 7 ⟩
  procedure initialize; { this procedure gets things started properly }
    var ⟨ Local variables for initialization 6 ⟩
    begin ⟨ Set initial values of key variables 8 ⟩
    end;
```

3. 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 }
```

4. The character set. (The following material is copied verbatim from TEX82. Thus, the same system-dependent changes should be made to both programs.)

In order to make TEX readily portable between a wide variety of computers, all of its input text is converted to an internal seven-bit code that is essentially standard ASCII, the "American Standard Code for Information Interchange." This conversion is done immediately when each character is read in. Conversely, characters are converted from ASCII to the user's external representation just before they are output to a text file.

Such an internal code is relevant to users of TEX primarily because it governs the positions of characters in the fonts. For example, the character 'A' has ASCII code 65 = '101, and when TEX typesets this letter it specifies character number 65 in the current font. If that font actually has 'A' in a different position, TEX doesn't know what the real position is; the program that does the actual printing from TEX's device-independent files is responsible for converting from ASCII to a particular font encoding.

TEX's internal code is relevant also with respect to constants that begin with a reverse apostrophe; and it provides an index to the \catcode, \mathcode, \uccode, \lccode, and \delcode tables.

5. Characters of text that have been converted to TEX's internal form are said to be of type *ASCII_code*, which is a subrange of the integers.

⟨ Types in the outer block 5 ⟩ ≡
```
ASCII_code = 0 .. 127; { seven-bit numbers }
```
This code is used in section 2.

6. The original Pascal compiler was designed in the late 60s, when six-bit character sets were common, so it did not make provision for lower case letters. Nowadays, of course, we need to deal with both upper and lower case alphabets in a convenient way, especially in a program for typesetting; so the present specification of TEX has been written under the assumption that the Pascal compiler and run-time system permit the use of text files with more than 64 distinguishable characters. More precisely, we assume that the character set contains at least the letters and symbols associated with ASCII codes '40 through '176; all of these characters are now available on most computer terminals.

Since we are dealing with more characters than were present in the first Pascal compilers, we have to decide what to call the associated data type. Some Pascals use the original name **char** for the characters in text files, even though there now are more than 64 such characters, while other Pascals consider **char** to be a 64-element subrange of a larger data type that has some other name.

In order to accommodate this difference, we shall use the name **text_char** to stand for the data type of the characters that are converted to and from **ASCII_code** when they are input and output. We shall also assume that **text_char** consists of the elements **chr(first_text_char)** through **chr(last_text_char)**, inclusive. The following definitions should be adjusted if necessary.

```
define text_char ≡ char { the data type of characters in text files }
define first_text_char = 0 { ordinal number of the smallest element of text_char }
define last_text_char = 127 { ordinal number of the largest element of text_char }
```

⟨ Local variables for initialization 6 ⟩ ≡
```
i: 0 .. last_text_char;
```
This code is used in section 2.

7. The TEX processor converts between ASCII code and the user's external character set by means of arrays **xord** and **xchr** that are analogous to Pascal's **ord** and **chr** functions.

⟨ Globals in the outer block 7 ⟩ ≡
```
xord: array [text_char] of ASCII_code; { specifies conversion of input characters }
xchr: array [ASCII_code] of text_char; { specifies conversion of output characters }
```
See also sections 12, 13, and 18.

This code is used in section 2.

8. Since we are assuming that our Pascal system is able to read and write the visible characters of standard ASCII (although not necessarily using the ASCII codes to represent them), the following assignment statements initialize most of the *xchr* array properly, without needing any system-dependent changes. On the other hand, it is possible to implement TEX with less complete character sets, and in such cases it will be necessary to change something here.

⟨ Set initial values of key variables 8 ⟩≡

```
xchr['40] ← ' '; xchr['41] ← '!'; xchr['42] ← '"'; xchr['43] ← '#'; xchr['44] ← '$';
xchr['45] ← '%'; xchr['46] ← '&'; xchr['47] ← '''';
xchr['50] ← '('; xchr['51] ← ')'; xchr['52] ← '*'; xchr['53] ← '+'; xchr['54] ← ',';
xchr['55] ← '-'; xchr['56] ← '.'; xchr['57] ← '/';
xchr['60] ← '0'; xchr['61] ← '1'; xchr['62] ← '2'; xchr['63] ← '3'; xchr['64] ← '4';
xchr['65] ← '5'; xchr['66] ← '6'; xchr['67] ← '7';
xchr['70] ← '8'; xchr['71] ← '9'; xchr['72] ← ':'; xchr['73] ← ';'; xchr['74] ← '<';
xchr['75] ← '='; xchr['76] ← '>'; xchr['77] ← '?';
xchr['100] ← '@'; xchr['101] ← 'A'; xchr['102] ← 'B'; xchr['103] ← 'C'; xchr['104] ← 'D';
xchr['105] ← 'E'; xchr['106] ← 'F'; xchr['107] ← 'G';
xchr['110] ← 'H'; xchr['111] ← 'I'; xchr['112] ← 'J'; xchr['113] ← 'K'; xchr['114] ← 'L';
xchr['115] ← 'M'; xchr['116] ← 'N'; xchr['117] ← 'O';
xchr['120] ← 'P'; xchr['121] ← 'Q'; xchr['122] ← 'R'; xchr['123] ← 'S'; xchr['124] ← 'T';
xchr['125] ← 'U'; xchr['126] ← 'V'; xchr['127] ← 'W';
xchr['130] ← 'X'; xchr['131] ← 'Y'; xchr['132] ← 'Z'; xchr['133] ← '['; xchr['134] ← '\';
xchr['135] ← ']'; xchr['136] ← '^'; xchr['137] ← '_';
xchr['140] ← '`'; xchr['141] ← 'a'; xchr['142] ← 'b'; xchr['143] ← 'c'; xchr['144] ← 'd';
xchr['145] ← 'e'; xchr['146] ← 'f'; xchr['147] ← 'g';
xchr['150] ← 'h'; xchr['151] ← 'i'; xchr['152] ← 'j'; xchr['153] ← 'k'; xchr['154] ← 'l';
xchr['155] ← 'm'; xchr['156] ← 'n'; xchr['157] ← 'o';
xchr['160] ← 'p'; xchr['161] ← 'q'; xchr['162] ← 'r'; xchr['163] ← 's'; xchr['164] ← 't';
xchr['165] ← 'u'; xchr['166] ← 'v'; xchr['167] ← 'w';
xchr['170] ← 'x'; xchr['171] ← 'y'; xchr['172] ← 'z'; xchr['173] ← '{'; xchr['174] ← '|';
xchr['175] ← '}'; xchr['176] ← '~';
xchr[0] ← ' '; xchr['177] ← ' '; { ASCII codes 0 and '177 do not appear in text }
```

See also sections 10, 11, and 14.

This code is used in section 2.

9. Some of the ASCII codes without visible characters have been given symbolic names in this program because they are used with a special meaning.

```
define null_code = '0   { ASCII code that might disappear }
define carriage_return = '15   { ASCII code used at end of line }
define invalid_code = '177   { ASCII code that should not appear }
```

10. The ASCII code is "standard" only to a certain extent, since many computer installations have found it advantageous to have ready access to more than 94 printing characters. Appendix C of The TEXbook gives a complete specification of the intended correspondence between characters and TEX's internal representation.

If TEX is being used on a garden-variety Pascal for which only standard ASCII codes will appear in the input and output files, it doesn't really matter what codes are specified in xchr[1 . . '37], but the safest policy is to blank everything out by using the code shown below.

However, other settings of ***xchr*** will make TEX more friendly on computers that have an extended character set, so that users can type things like '≠' instead of '\ne'. At MIT, for example, it would be more appropriate to substitute the code

for $i \leftarrow 1$ to 37 do ***xchr [i] ← chr (i);***

TEX's character set is essentially the same as MIT's, even with respect to characters less than ***'40.*** People with extended character sets can assign codes arbitrarily, giving an ***xchr*** equivalent to whatever characters the users of TEX are allowed to have in their input files. It is best to make the codes correspond to the intended interpretations as shown in Appendix C whenever possible; but this is not necessary. For example, in countries with an alphabet of more than 26 letters, it is usually best to map the additional letters into codes less than '40.

⟨ Set initial values of key variables 8 ⟩ +≡

for $i \leftarrow 1$ **to 37 do** $xchr[i] \leftarrow$ '␣';

11. The following system-independent code makes the ***xord*** array contain a suitable inverse to the information in ***xchr.*** Note that if $xchr[i] = xchr[j]$ where $i < j < 177$, the value of $xord[xchr[i]]$ will turn out to be j or more; hence, standard ASCII code numbers will be used instead of codes below '40 in case there is a coincidence.

⟨ Set initial values of key variables 8 ⟩ +≡

for *i* ← *first-text-char* **to** *In&text-char* **do** *xord* [$chr(i)$] ← *invalid_code*;

for *i* ← 1 **to 176 do** *xord [xchr [i]]* ← *i;*

12. String handling. (The following material is copied from the *init_strings* procedure of TEX82, with slight changes.)

⟨ Globals in the outer block 7 ⟩ +≡
```
k, l: 0 . . 127;  { small indices or counters }
m, n: text-char;  { characters input from pool_file }
s: integer;  { number of strings treated so far}
```

13. The global variable ***count*** keeps track of the total number of characters in strings.

⟨ Globals in the outer block 7 ⟩ +≡
```
count: integer;  { how long the string pool is, so far }
```

14. ⟨ Set initial values of key variables 8⟩ +≡
```
count ← 0;
```

15. This is the main program, where POOLtype starts and ends.
```
define abort (#) ≡
        begin write-Zn(#); got0 9999;
        end
begin initialize;
⟨ Make the first 128 strings 16 ⟩;
s ← 128;
⟨ Read the other strings from the POOL file, or give an error message and abort 19 ⟩;
write-Zn ('(', count : 1, ' characters in all.)');
9999: end.
```

16. ⟨ Make the first 128 strings 16⟩ ≡
```
for k ← 0 to 127 do
  begin write (k : 3, ': "');  l ← k;
  if (⟨ Character k cannot be printed 17 ⟩) then
    begin if k < '100 then l + k + '100 else l ← k − '100;
    write(xchr["^"], xchr["^"]);  count ← count + 2;
    end;
  if l = "!" then write(xchr[l], xchr[l])
  else write (xchr[l]);
  incr(count);  write-Zn( '"' );
  end
```
This code is used in section 15.

17. The first 128 strings will contain 95 standard ASCII characters, and the other 33 characters will be printed in three-symbol form like '^^A' unless a system-dependent change is made here. Installations that have an extended character set, where for example *xchr*['32] = '≠', would like string '32 to be the single character '32 instead of the three characters '136, '136, '132 (^^Z). On the other hand, even people with an extended character set will want to represent string '15 by ^^M, since '15 is *carriage-return;* the idea is to produce visible strings instead of tabs or line-feeds or carriage-returns or bell-rings or characters that are treated anomalously in text files.

The boolean expression defined here should be *true* unless TEX internal code number *k* corresponds to a non-troublesome visible symbol in the local character set. At MIT, for example, the appropriate formula would be '$k \in$ [0, '10 .. '12, '14, '15, '33, '177]'. If character *k* cannot be printed, then character *k* + '100 or *k* − '100 must be printable; thus, at least 64 printable characters are needed.

⟨ Character *k* cannot be printed 17 ⟩ ≡

```
(k < "␣") ∨ (k > "~")
```

This code is used in section 16.

18. When the **WEB** system program called TANGLE processes a source file, it outputs a Pascal program and also a string pool file. The present program reads the latter file, where each string appears as a two-digit decimal length followed by the string itself, and the information is output with its associated index number. The strings are surrounded by double-quote marks; double-quotes in the string itself are repeated.

⟨ Globals in the outer block 7 ⟩ +≡

```
pool-file: packed file of text-char;  { the string-pool file output by TANGLE }
xsum : boolean ;  { has the check sum been found? }
```

19. ⟨ Read the other strings from the **POOL** file, or give an error message and abort 19 ⟩ ≡

```
reset (pool-file): xsum ← false;
if eof (pool-file) then abort('!␣I␣can''t␣read␣the␣POOL␣file.');
repeat ⟨ Read one string, but abort if there are problems 20⟩;
until xsum;
if ¬eof (pool-file) then abort ('!␣There''s␣junk␣after␣the␣check␣sum')
```

This code is used in section 15.

20. ⟨ Read one string, but abort if there are problems 20⟩ ≡

```
if eof (pool-file) then abort('!␣POOL␣file␣contained␣no␣check␣sum');
read(pool_file, m, n);  { read two digits of string length }
if m ≠ '*' then
  begin if (xord[m] < "0") ∨ (xord[m] > "9") ∨ (xord[n] < "0") ∨ (xord[n] > "9") then
    abort('!␣POOL␣line␣doesn''t␣begin␣with␣two␣digits');
  l ← xord[m] * 10 + xord[n] − "0" * 11;  {compute the length}
  write (s : 3, ' :␣"'); count ← count + l;
  for k ← 1 to l do
    begin if eoln (pool-file) then
      begin write-Zn ('"'); abort ('!␣That␣POOL␣line␣was␣too␣short');
      end;
    read(pool_file, m); write(xchr[xord[m]]);
    if xord[m] = """" then write (xchr["""]);
    end;
  write-Zn ('"'); incr (s);
  end
else xsum ← true;
read-Zn (pool-file)
```

This code is used in section 19.

21. System-dependent changes. This section should be replaced, if necessary, by changes to the program that are necessary to make **POOLtype** 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.

22. Index. Indications of system dependencies appear here together with the section numbers where each identifier is used.

The TFtoPL processor

(Version 2.5, September 1985)

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.

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

define ***banner*** ≡ 'This␣is␣TFtoPL,␣Version␣2.5' { 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-h***(#) ≡ ***write-Zn (X)***

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 parameter can be changed at compile time to extend or reduce TFtoPL's capacity.

(Constants in the outer block 4) ≡
tfm-size = **20000**; { maximum length of ***tfm*** data, in bytes }

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, and 68.
This code is used in section 2.

7. On some systerns 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*** ≤ 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) +≡
Zf. Zh, 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

header : **array** [**0** . . *lh* − 1] **of** *stuff*
char-info : **array** *[bc . . ec]* **of** *char-info-word*
width : **array** [**0** . . *nw* − 1] **of** *fix_word*
height : **array** [**0** . . *nh* − 1] **of** *fix-word*
depth : **array** [**0** . . *nd* − 1] **of** *fix-word*
italic : **array** [**0** . . *ni* − 1] **of** *fix-word*
Zig-kern : **array** [**0** . . *nl* − 1] **of** *Zig-kern-command*
kern : **array** [**0** . . *nk* − 1] **of** *fix-word*
exten : **array** [**0** . . *ne* − 1] **of** *extensible-recipe*
param : **array** [1 . . *np]* **of** *fix-word*

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[l] 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 lo-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 6, 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[l]*** 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.) Oriental fonts, for which many different individual symbols might share the same metric information, should be identifiable via this part of the TFM header.

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 ***tug*** (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 ***tug*** field in a *char_info_word* has four values that explain how to interpret the ***remainder*** field.

tug = 0 (no-tug) means that ***remuinder*** is unused.
tug = 1 ***(Zig-tug)*** means that this character has a ligature/kerning program starting at ***Zig-kern[remuinder].***
tug = 2 (list-tug) means that this character is part of a chain of characters of ascending sizes, and not the largest in the chain. The ***remninder*** field gives the character code of the next larger character.
tug = 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-tug*** = 0 { vanilla character }
define ***Zig-tug*** = 1 { character has a ligature/kerning program}
define ***list-tug*** = 2 {character has a successor in a charlist}
define *ext_tag* = 3 {character is extensible}

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

first byte: ***stop-bit,*** indicates that this is the final program step if the byte is 128 or more.
second byte: ***next-char,*** "if ***next-char*** follows the current character, then perform the operation and stop, otherwise continue."
third byte: *op_bit*, indicates a ligature step if less than 128. a kern step otherwise.
fourth byte: ***remuinder .***

In a ligature step the current character and ***next-chur*** are replaced by the single character whose code is ***remainder.*** In a kern step, an additional space equal to ***kern [remuinder]*** is inserted between the current character and ***next-chur.*** (The value of *kern*[*remainder*] is often negative, so that the characters are brought closer together by kerning: but it might be positive.)

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-purum*** array, which is another sequence of ***fix-word*** values.

purum [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.

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

purum [7] = ***extru-space*** is the amount added to ***purum*** [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*, ***denoml***, *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 ● tUgoUon;UareUyouUsureUthisUis,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_Zn('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 85.

21. Once 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␣subf ile␣sizes␣is␣negative!');
    # ← tjm[tjm-ptr] * 400 + tjm[tjm-ptr + 1]; tjm-ptr ← tjm-ptr + 2;
    end;
```

(Set subfile sizes *lh*, bc, ... *np* 21 ⟩ ≡

```
begin tjm-ptr ← 2;
eval_two_bytes(Zh); 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 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!');
if (nw = 0) ∨ (nh = 0) ∨ (nd = 0) ∨ (ni = 0) then
  abort('Incomplete␣subfiles␣for␣character␣dimensions!');
if, (bc > ec + 1) ∨ (ec > 255) then
  abort('The␣character␣code␣range␣', bc : 1, '..', ec : 1, '␣is␣illegal!');
if ne > 256 then abort('There␣are␣', ne : 1, '␣extensible␣recipes!');
end
```

This code is used in section 85.

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 *tjm* array.

(Globals in the outer block 6 ⟩ +≡

```
char-base, width-base, height-base, depth-base, italic-base, Zig-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; Zig-kern-base ← italic-base + ni;
kern-base ← Zig-kern-base + nl; exten-base ← kern-base + nk; param_base ← exten-base + ne − 1;
end
```

This code is used in section 85.

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 ***tjm [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*** (#) ≡ ***(tjm [char-info*** (#) + ***I]*** **mod 16)**
define ***italic-index*** (#) ≡ ***(tjm*** [*char_info*(#) + 2] **div 4)**
define *tag* (#) ≡ ***(tjm*** [*char_info* (#) + 2] mod 4)
define ***reset-tag(#)*** ≡ ***tjm [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 ***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-U4. 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 $dig[j-1] \ldots dig[0]$, 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 $l \leq 4$, 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 }
  vara: 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 + Z];
  2: a ← a + 4 * tfm[k + Z];
  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-octuZ(0, 1)
    end
  else if (c ≥ "0") A (c ≤ "9") then out('␣C␣', c − "0" : 1)
    else if (c ≥ "A") A (c ≤ "Z") then out('␣C␣', ASCII_10[c − "A" + 2])
      else if (c ≥ "a") 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 inforrriation 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 justs 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) +≡

```
churs-on-line ← 0;
perfect ← true;  { innocent until proved guilty }
```

47. Error messages are given with the help of the *bud* and *range-error* and *bud-char* macros:

```
define bud (#) ≡
        begin perfect ← false;
        if chars-on-line > 0 then print_ln(´␣´);
        chars_on_line ← 0; print-Zn (´Bad␣TFM␣file␣:␣´, #);
        end
define range-error (#) ≡
        begin perfect ← false; print-Zn(´␣´); print (#, ´␣index␣f or␣character␣´); print-octal(c);
        print-Zn(´␣is␣too␣large;´); print-Zn(´so␣I␣reset␣it␣to␣zero.´);
        end
define bud-chur-taiZ(#) ≡ print_octal(#); print-Zn (´.´);
      end
define bud-chur (#) ≡
      begin perfect ← false;
      if chars-on-line > 0 then print-Zn(´␣´);
      chars_on_line ← 0; print (´Bad␣TFM␣file␣:␣´, #, ´␣nonexistent␣character␣´); bud-char-tail
```

```
( Globals in the outer block 6) +≡
i: 0 . . 77777:  { an index to words of a subfile }
c, r: byte:  { random characters }
k: index;  { a random index }
```

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 86.

49. ⟨ Output the check sum 49 ⟩ ≡

```
  left; out (´CHECKSUM´);
  if lh = 0 then out (´␣0␣0´) else out_octal(check-sum, 4);
  right
```

This code is used in section 48.

50. Incorrect design sizes are changed to 10 points.

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

51. ⟨ Output the design size 51 ⟩ ≡

```
  left; out (´DESIGNSIZE´);
  if lh < 2 then bad_design( 'missing')
  else if tfm [design-size] > 127 then bud-design ( 'negative´)
    else if (tfm[design_size] = 0) A (tfm [design-size + 1] < 16) then bad_design(´too␣small´)
      else out_fix (design-size);
  right; out (´(COMMENT␣DESIGNSIZE␣IS␣IN␣POINTS)´); out-Zn;
  out(´(COMMENT␣OTHER␣SIZES␣ARE␣MULTIPLES␣OF␣DESIGNSIZE)´); out-Zn
```

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 = "(") V (c = ")") then
      begin bad('Parenthesis␣in␣string␣has␣been␣changed␣to␣slash.'); tfm[j] ← "/";
      end
    else if (c < "␣") V (c > "~") then
        begin bad('Nonstandard␣ASCII␣code␣has␣been␣blotted␣out.'); tfm[j] ← "?";
        end
      else if (c ≥ "a") 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") A
       (tfm [scheme + 4] = "␣") A ( tfm [scheme + 5] = "M") A (tfm [scheme + 6] = "A") ∧
       (tfm[scheme + 7] = "T") ∧ (tfm[scheme + 8] = "H") ∧ (tfm[scheme + 9] = "␣") then
    begin if (tfm[scheme + 10] = "S") A (tfm[scheme + 11] = "Y") then font-type ← mathsy
    else if (tfm[scheme + 10] = "E") A (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-octaZ( 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 (Zh > 17) A (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 86.

59. ⟨ Check to see if *np* is complete for this font type 59 ⟩ ≡

```
  if (font-type = mathsy) A (np ≠ 22) then
    print_ln('Unusual number of fontdimen parameters for a math symbols font (', np : 1,
        ' not 22).')
  else if (font-type = mathex) A (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) A (tfm[#] < 255) then
         begin tfm[#] ← 0: tfm[(#) + 1] ← 0; tfm[(#) + 2] ← 0; tfm[(#) + 3] ← 0: check_fix_tail
```

⟨ Check and output the *i*th 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: ouf('SPACE'); 3: out('STRETCH'); 4: out(-SHRINK');
  5: out(´XHEIGHT´); 6: out(-QUAD'); 7: out(´EXTRASPACE´)
  end
else if (i ≤ 22) A (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) A (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) V (tfm[# + 1] > 0) V (tfm [# + 2] > 0) V (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 86.

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 ***Zig-tag*** and whose ***remainder*** is *r*, we will store the pair ***(c, r)*** in the ***label-table*** array. 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 . . 257] of record
    cc: byte:
    rr: 0 . . 256:
    end:
label_ptr: 0 . . 256:   { the largest entry in label_table }
sort-ptr: 0 . . 256:    { index into label_table }
```

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

65. ⟨ Do the ligatures and kerns 65 ⟩≡
```
⟨ Build the label table 66 ⟩;
if nl > 0 then
  begin left; out ( 'LIGTABLE' ); out_ln;
  ⟨ Output the ligature/kern program 69 ⟩;
  right;
  end
```
This code is used in section 88.

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

⟨ Build the label table 66 ⟩≡
```
for c ← bc to ec do
  if tag(c) = Zig-tag then
    begin r ← remainder(c);
    if r ≥ nl then
      begin range-error ( 'Ligature/kern' ); reset-tag(c);
      end
    else ( Insert (c, r) into label-table 67);
    end;
label_table[label_ptr + 1].rr ← 256;  { put "infinite" sentinel at the end }
```
This code is used in section 65.

67. ⟨ Insert (c, r) into label-table 67 ⟩≡
```
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 );
end
```
This code is used in section 66.

68. As we translate the ligature/kern program into symbolic form, we will keep track of whether or not the program steps are actually accessible from some character.

⟨ Globals in the outer block 6) +≡
```
active : boolean  { is there a way to get to the present step? }
```

69. When '(STOP)' is output on level 2, an inaccessible portion of the ligature/kern program that is *being* commented out has just ended, so we want to emit an extra right parenthesis.

```
define out-stop ≡
        begin out ('(STOP)'); out_ln;
        if level > 1 then right;
        end
⟨ Output the ligature/kern program 69) ≡
  active ← false; sort-ptr ← 1;
  for i ← 0 to nl − 1 do
    begin ⟨ Output any labels for step i 70);
    if ¬active then ⟨ Output a comment about the redundancy 71 ⟩;
    ⟨ Output step i of the ligature/kern program 72);
    end;
  if active then
    begin bad('No␣stop␣bit␣at␣the␣end␣of␣ligature/kern␣program.'); out-stop;
    tfm[kern(0) − 4) ← tfm[kern(0) − 4] + stop_flag;
    end
```

This code is used in section 65.

70. ⟨ Output any labels for step i 70) ≡

```
while i = label_table[sort_ptr].rr do
  begin if level > 1 then right;
  active ← true; left; out ('LABEL'); out_char(label_table[sort_ptr].cc); right; incr(sort_ptr);
  end
```

This code is used in section 69.

71. ⟨ Output a comment about the redundancy 71 ⟩ ≡

```
begin left; out ('COMMENT␣THIS␣PART␣OF␣THE␣PROGRAM␣IS␣NEVER␣USED!'); out-Zn; active ← true;
    { the right parenthesis will be emitted by out-stop or by the next label output }
end
```

This code is used in section 69.

72. ⟨ Output step i of the ligature/kern program 72) ≡

```
begin k ← 4 * (lig_kern_base + i);
if tfm[k + 2] ≥ kern_flag then ⟨ Output a kern step 73 ⟩
else ⟨ Output a ligature step 74);
if tfm[k] ≥ stop-flag then
  begin if sort-ptr > 0 then out-stop:
  active ← false;
  end;
end
```

This code is used in sections 69 and 81.

73. ⟨ Output a kern step 73 ⟩ ≡

```
begin if nonexistent(tfm[k + 1]) then bad-char(´Kern␣step␣for´)(tfm[k + 1])
else begin left: out(´KRN´); out-char(tfm[k + 1]):
  if tfm[k + 3] ≥ nk then
    begin bad(´Kern␣index␣too␣large.´); out(´␣R␣0.0´);
    end
  else out_fix (kern (tfm [k + 3]));
  right;
  end;
end
```

This code is used in section 72.

74. ⟨ Output a ligature step 74) 3

```
begin if nonexistent (tfm [k + 1]) then bad-char(´Ligature␣step␣for´)(tfm[k + 1]);
if nonexistent (tfm [k + 3]) then bad_char(´Ligature␣step␣produces␣the´)(tfm[k + 3])
else begin left: out (´LIG´): out_char(tfm[k + 1]); out_char(tfm[k + 3]); right;
  end:
end
```

This code is used in section 72.

75. 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 75 ⟩ ≡

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

This code is used in section 88.

76. 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 76 ⟩ ≡
  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-Zn (' '); 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-Zn; ⟨ Output the character's width 77 ⟩;
      if height-in&x(c) > 0 then ⟨ Output the character's height 78);
      if depth-index(c) > 0 then ⟨ Output the character's depth 79 ⟩;
      if italic-index(c) > 0 then ⟨ Output the italic correction 80);
      case tug(c) of
      no-tag: do-nothing;
      lig_tag: ⟨ Output the applicable part of the ligature/kern program as a comment 81 ⟩;
      list-tag: ⟨ Output the character link unless there is a problem 82);
      ext_tag: ⟨ Output an extensible character recipe 83):
      end; right;
      end
```

This code is used in section 87.

77. ⟨ Output the character's width 77 ⟩ ≡

```
  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 76.

78. ⟨ Output the character's height 78) ≡

```
  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 76.

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

```
  if depth-index(~) ≥ nd then range-error ('Depth')
  else begin left : out ('CHARDP'): out-fix (depth(c)): right;
    end
```

This code is used in section 76.

80. ⟨ Output the italic correction 80) ≡

```
  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 76.

81. ⟨ Output the applicable part of the ligature/kern program as a comment 81) ≡

```
begin left; out ('COMMENT'); out,Zn;
i ← remainder(c); active ← true;
repeat ⟨ Output step i of the ligature/kern program 72);
  incr(i);
until active = false;
right;
end
```

This code is used in section 76.

82. We want to make sure that there is no cycle of characters linked together by *list-tag* entries, since such a cycle would get TEX into an endless loop. 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 82) ≡

```
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) A (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-Zn('␣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 76.

83. ⟨ Output an extensible character recipe 83) ≡

```
if remainder(c) ≥ ne then
  begin range_error('Extensible'); reset-tag(c);
  end
else begin left; out('VARCHAR'); out-Zn; ⟨ Output the extensible pieces that exist 84);
  right;
  end
```

This code is used in section 76.

84. ⟨ Output the extensible pieces that exist 84) ≡

```
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 83.

85. **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 Zh, bc, . . . , np 21⟩;
  ⟨ Compute the base addresses 23 ⟩;
  organize ← true; got0 30;
final-end: organize ← false;
30: end;
```

86. Next we do the simple things.

```
procedure do-simple-things;
  vari: 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:
```

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

```
procedure do-characters;
  var c: byte;  { character being done }
    k: index;  { a random index}
  begin ⟨ Do the characters 76⟩;
  end;
```

88. Here is where TFtoPL begins and ends.

```
  begin initialize;
  if ¬organize then goto final-end;
 do-simple-things ;
  ⟨ Do the ligatures and kerns 65 ⟩;
  ⟨ Check the extensible recipes 75 ⟩;
  do-characters; print-Zn( '.');
  if level ≠ 0 then print-Zn('This␣program␣isn''t␣working!');
  if ¬perfect then out ( '(COMMENT␣THE␣TFM␣FILE␣WAS␣BAD,␣SO␣THE␣DATA␣HAS␣BEEN␣CHANGED!)');
"final-end: end.
```

89. 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.

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

The PLtoTF processor

(Version 2.3, August 1985)

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.

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

```
define banner ≡ 'This␣is␣PLtoTF,␣Version␣2.3'  { 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(#)
define print_ln(#) ≡ write_ln(#)
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)
  procedure initialize ; { this procedure gets things started properly }
    var ( Local variables for initialization 19 ⟩
    begin print-Zn (banner);
    ( 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 = 60;  { length of lines displayed in error messages }
  max_header_bytes = 100;  { four times the maximum number of words allowed in the TFM file header
      block, must be 1024 or less }
  max_param_words = 30:  { the maximum number of f ontdimen parameters allowed }
```

This code is used in section 2.

4. 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 }
```

5. Property list description of font metric data. The idea behind PL files is that precise details about fonts, i.e., the facts that are needed by typesetting routines like TEX, sometimes have to be supplied by hand. The nested property-list format provides a reasonably convenient way to do this.

A good deal of computation is necessary to parse and process a PL file, so it would be inappropriate for TEX itself to do this every time it loads a font. TEX deals only with the compact descriptions of font metric data that appear in TFM files. Such data is so compact, however, it is almost impossible for anybody but a computer to read it. The purpose of PLtoTF is to convert from a human-oriented file of text to a computer-oriented file of binary numbers.

⟨ Globals in the outer block 5 ⟩ ≡
```
pl_file : text ;
```
See also sections 15, 18, 21, 23, 25, 30, 36, 38, 39, 44, 58, 65, 67, 72, 76, 79, 81, 99, 108, 112, 117, 121, and 124.
This code is used in section 2.

6. ⟨ Set initial values 6) ≡
```
reset (pl_file);
```
See also sections 16, 20, 22, 24, 26, 37, 41, 70, and 74.
This code is used in section 2.

7. A PL file is a list of entries of the form

```
(PROPERTYNAME VALUE)
```

where the property name is one of a finite set of names understood by this program, and the value may itself in turn be a property list. The idea is best understood by looking at an example, so let's consider a fragment of the PL file for a hypothetical font.

```
(FAMILY NOVA)
(FACE F MIE)
(CODINGSCHEME ASCII)
(DESIGNSIZE D 10)
(DESIGNUNITS D 18)
(COMMENT A COMMENT IS IGNORED)
(COMMENT (EXCEPT THIS ONE ISN'T))
(COMMENT (ACTUALLY IT IS, EVEN THOUGH
        IT SAYS IT ISN'T))
(FONTDIMEN .
    (SLANT R -.25)
    (SPACE D 6)
    (SHRINK D 2)
    (STRETCH D 3)
    (XHEIGHT R 10.55)
    (QUAD D 18)

(LIGTABLE
    (LABEL C f)
    (LIG C i O 200)
    (LIG C f 0 201)
    (KRN 0 51 R 1.5)
    (STOP)
    (LABEL 0 201)
    (LIG C i 0 203)
    (STOP)

(CHARACTER C f
    (CHARWD D 6)
    (CHARHT R 13.5)
    (CHARIC R 1.5)
```

This example says that the font whose metric information is being described belongs to the hypothetical NOVA family: its face code is medium italic extended; and the characters appear in ASCII code positions. The design size is 10 points. and all other sizes in this PL file are given in units such that, 18 units equals the design size. The font is slanted with a slope of –.25 (hence the letters actually slant backward-perhaps that is why the family name is NOVA). The normal space between words is 6 units (i.e.. one third of the 18-unit design size). with glue that shrinks by 2 units or stretches by 3. The letters for which accents don't need to be raised or lowered are 10.55 units high, and one em equals 18 units.

The example ligature table specifies that the letter f followed by i is changed to code 'ZOO. while f followed by f is changed to *'201*; and if f is followed by the code '51 (which is a right parenthesis) an additional 1.5 units of space should be inserted after the f. The character code *'201* is changed to '203 if it is followed by i; thus. the sequence f f i leads to code '203. which is presumably where the 'ffi' ligature appears in the font.

Character f itself is 6 units wide and 13.5 units tall, in this example. Its depth is zero (since CHARDP is not given), and its italic correction is 1.5 units.

8. The example above illustrates most of the features found in PL files. Note that some property names, like FAMILY or COMMENT, take a string as their value: this string continues until the first unmatched right parenthesis. But most property names, like DESIGNSIZE and SLANT and LABEL, take a number as their value. This number can be expressed in a variety of ways, indicated by a prefixed code; D stands for decimal, H for hexadecimal, 0 for octal, R for real, C for character, and F for "face." Other property names, like LIG, take two numbers as their value. And still other names, like FONTDIMEN and LIGTABLE and CHARACTER, have more complicated values that involve property lists.

A property name is supposed to be used only in an appropriate property list. For example, CHARWD shouldn't occur on the outer level or within FONTDIMEN.

The individual property-and-value pairs in a property list can appear in any order. For instance, 'SHRINK' precedes STRETCH' in the above example, although the TFM file always puts the stretch parameter first. One could even give the information about characters like f ' before specifying the number of units in the design size, or before specifying the ligature and kerning table. However, the LIGTABLE itself is an exception to this rule; the individual elements of the LIGTABLE property list can be reordered only to a certain extent without changing the meaning of that table.

If property-and-value pairs are omitted. a default value is used. For example, we have already noted that the default for CHARDP is zero. The default for ever-y numeric value is, in fact, zero, unless otherwise stated below.

If the same property name is used more than once, PLtoTF will not notice the discrepancy; it simply uses the final value given. Once again, however. the LIGTABLE is an exception to this rule: PLtoTF will complain if there is more than one label for some character. And of course many of the entries in the LIGTABLE property list have the same property name.

From these rules, you can guess (correctly) that PLtoTF operates in four main steps. First it assigns the default values to all properties; then it scans through the PL file, changing property values as new ones are seen; then it checks the information and corrects any problems; and finally it outputs the TFM file.

9. Instead of relying on a hypothetical example, let's consider a complete grammar for PL files. At the outer level, the following property names are valid:

CHECKSUM (four-byte value). The value, which should be a nonnegative integer less than 2^{32}, is used to identify a particular version of a font; it should match the check sum value stored with the font itself. A check sum of zero, which is the default, is used to bypass check sum testing. If no checksum is specified in the PL file, PLtoTF will compute the checksum that METAFONT would compute from the same data.

DESIGNSIZE (numeric value, default is 10). The value, which should be a real number in the range $1.0 \leq x < 1024$, represents the default amount by which all quantities will be scaled if the font is not loaded with an 'at' specification. For example, if one says '\font A=cmr10 at 15pt' in TEX language, the design size in the TFM file is ignored and effectively replaced by 15 points; but if one simply says '\font A=cmr10' the stated design size is used. This quantity is always in units of printer's points.

DESIGNUNITS (numeric value, default is 1). The value should be a positive real number; it says how many units equals the design size (or the eventual 'at' size, if the font is being scaled). For example, suppose you have a font that has been digitized with 600 pixels per em, and the design size is one em; then you could say '(DESIGNUNITS D 600)' if you wanted to give all of your measurements in units of pixels.

CODINGSCHEME (string value, default is 'UNSPECIFIED'). The string should not contain parentheses, and its length must be less than 40. It identifies the correspondence between the numeric codes and font characters. (TEX ignores this information, but other software programs make use of it.)

FAMILY (string value, default is 'UNSPECIFIED'). The string should not contain parentheses, and its length must be less than 20. It identifies the name of the family to which this font belongs, e.g., 'HELVETICA'. (TEX ignores this information; but it is needed, for example, when converting DVI files to PRESS files for Xerox equipment.)

FACE (one-byte value). This number, which must lie between 0 and 255 inclusive, is a subsidiary identification of the font within its family. For example, bold italic condensed fonts might have the same family name as light roman extended fonts. differing only in their face byte. (TEX ignores this information; but it is needed, for example, when converting DVI files to PRESS files for Xerox equipment.)

SEVENBITSAFEFLAG (string value, default is 'FALSE'). The value should start with either 'T' (true) or 'F' (false). If true, character codes less than 128 cannot lead to codes of 128 or more via ligatures or charlists or extensible characters. (TEX82 ignores this flag, but older versions of TEX would only accept TFM files that were seven-bit safe.) PLtoTF computes the correct value of this flag and gives an error message only if a claimed "true" value is incorrect.

HEADER (a one-byte value followed by a four-byte value). The one-byte value should be between 18 and a maximum limit that can be raised or lowered depending on the compile-time setting of *max_header_bytes*. The four-byte value goes into the header word whose index is the one-byte value; for example, to set ***header***[18] ← 1. one may write '(HEADER D 18 0 1)'. This notation is used for header information that is presently unnamed. (TEX ignores it.)

FONTDIMEN (property list value). See below for the names allowed in this property list.

LIGTABLE (property list value). See below for the rules about this special kind of property list.

CHARACTER. The value is a one-byte integer followed by a property list. The integer represents the number of a character that is present in the font; the property list of a character is defined below. The default is an empty property list.

10. Numeric property list values can be given in various forms identified by a prefixed letter.

C denotes an ASCII character, which should be a standard visible character that is not a parenthesis. The numeric value will therefore be between '41 and '176 but not '50 or '51.

D denotes a decimal integer, which must be nonnegative and less than 256. (Use R for larger values or for negative values.)

F denotes a three-letter Xerox face code; the admissible codes are MRR, MIR, BRR, BIR, LRR, LIR, MRC, MIC, BRC, BIC, LRC, LIC, MRE, MIE, BRE, BIE, LRE, and LIE, denoting the integers 0 to 17, respectively.

O denotes an unsigned octal integer, which must be less than 2^{32}, i.e., at most 'O 37777777777'.

H denotes an unsigned hexadecimal integer, which must be less than 2^{32}, i.e., at most 'H FFFFFFFF'.

R denotes a real number in decimal notation, optionally preceded by a '+' or '-' sign, and optionally including a decimal point. The absolute value must be less than 1024.

11. The property names allowed in a FONTDIMEN property list correspond to various TEX parameters, each, of which has a (real) numeric value. All of the parameters except SLANT are in design-size units. The admissible names are SLANT, SPACE, STRETCH, SHRINK, XHEIGHT, QUAD, EXTRASPACE, NUM1, NUM2, NUM3, DENOM1, DENOM2. SUP1, SUP2, SUP3, SUB1, SUB2, SUPDROP, SUBDROP, DELIM1, DELIM2, and AXISHEIGHT, for parameters 1 to 22. The alternate names DEFAULTRULETHICKNESS, BIGOPSPACING1, BIGOPSPACING2, BIGOPSPACING3, BIGOPSPACING4, and BIGOPSPACING5, may also be used for parameters 8 to 13.

The notation 'PARAMETER *n*' provides another way to specify the nth parameter; for example, '(PARAMETER D 1 R -.25)' is another way to specify that the SLANT is -0.25. The value of ***n*** must be positive and less than ***max-param-words.***

12. The elements of a CHARACTER property list can be of six different types.

CHARWD (real value) denotes the character's width in design-size units.

CHARHT (real value) denotes the character's height in design-size units.

CHARDP (real value) denotes the character's depth in design-size units.

CHARIC (real value) denotes the character's italic correction in design-size units.

NEXTLARGER (one-byte value), specifies the character that follows the present one in a "charlist." The value must be the number of a character in the font, and there must be no infinite cycles of supposedly larger and larger characters.

VARCHAR (property list value), specifies an extensible character. This option and NEXTLARGER are mutually exclusive: i.e., they cannot both be used within the same CHARACTER list.

The elements of a VARCHAR property list are either TOP, MID, BOT or REP; the values are integers, which must be zero or the number or a character in the font. A zero value for TOP, MID, or BOT means that the corresponding piece of the extensible character is absent. A nonzero value, or a REP value of zero, denotes the character code used to make up the top, middle, bottom, or replicated piece of an extensible character.

13. A LIGTABLE property list contains elements of four kinds, specifying a program in a simple command language that TEX uses for ligatures and kerns.

LABEL (one-byte value) means that the program for the stated character value starts here. The integer must be the number of a character in the font; its CHARACTER property list must not have a NEXTLARGER or VARCHAR field.

LIG (two one-byte values). The instruction '(LIG c *r*)' means, "If the next character is c, then replace both the current character and c by the character *r*; otherwise go on to the next instruction." Character *r* must be present. in the font. but c need not be.

KRN (a one-byte value and a real value). The instruction '(KRN c *r*)' means, "If the next character is *c*, then insert a blank space of width *r* between the current character character and *c*; otherwise go on to the next intruction." The value of *r*. which is in units of the design size, is often negative. Character code c must exist in the font.

STOP (no value). This instruction ends a ligature/kern program. It must follow either a LIG or KRN instruction, not a LABEL or STOP.

14. **In** addition to all these possibilities, the property name COMMENT is allowed in any property list. Such comments arc ignored.

15. So that is what PL files hold. The next question is, "What about TFM files?" A complete answer to that question appears in the documentation of the companion program, **TFtoPL**, so it will not be repeated here. Suffice it to say that a TFM file stores all of the relevant font information in a sequence of 8-bit bytes. The number of bytes is always a multiple of 4, so we could regard the TFM file as a sequence of 32-bit words; but TEX uses the byte interpretation, and so does PLtoTF. Note that the bytes are considered to be unsigned numbers.

(Globals in the outer block 5) +≡
tfm_file: packed file of 0 .. 255;

16. On some systems you may have to do something special to write 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 **6)** +≡
rewrite (*tfm_file*);

17. Basic input routines. For the purposes of this program, a byte is an unsigned eight-bit quantity, and an *ASCII_code* is an integer between '40 and '177. Such ASCII codes correspond to one-character constants like "A" in WEB language.

```
⟨ Types in the outer block 17 ⟩ ≡
  byte = 0 . . 255;  { unsigned eight-bit quantity }
  ASCII-code = '40 . . '177;  { standard ASCII code numbers }
```

See also sections 57, 61, 68, and 71.

This code is used in section 2.

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 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 }
```

19.
```
⟨ Local variables for initialization 19 ⟩ ≡
k: first-ord . . last-ord;  { an index used for clearing xord }
```

See also sections 40. 69, and 73.

This code is used in section 2.

20. Characters that should not appear in PL files (except in comments) are mapped into '177.

```
  define invalid-code = '177  { code deserving an error message }
⟨ Set initial values 6) +≡
  for k ← first-ord to last-ord do xord [ chr(k)] ← invalid-code;
  xord['␣'] ← "␣"; xord['!'] ← "!"; xord['"'] t """"; xord['#'] ← "#"; xord['$'] ← "$";
  xord['%'] ← "%"; xord['&'] ← "&"; xord['''] t "'"; xord['('] t "("; xord[')'] t ")";
  xord['*'] t "*"; xord['+'] t "+"; xord[','] t ","; xord['-'] t "-"; xord['.'] t ".";
  xord['/'] ← "/"; xord['0'] ← "0"; xord['1'] t "1"; xord['2'] t "2"; xord['3'] t "3";
  xord['4'] ← "4"; xord['5'] ← "5"; xord['6'] t "6"; xord['7'] t "7"; xord['8'] t "8";
  xord['9'] ← "9"; xord[':'] t ":"; xord[';'] ← ";"; xord['<'] ← "<"; xord['='] ← "=";
  xord['>'] ← ">"; xord['?'] t "?"; xord['@'] t "@"; xord['A'] ← "A"; xord['B'] t "B";
  xord['C'] ← "C"; xord[ D ] ← "D"; xord[ E ] ← "E"; xord [ 'F'] ← "F"; xord['G'] t "G";
  xord['H'] ← "H"; xord['I'] ← "I"; xord['J'] t "J"; xord['K'] t "K"; xord['L'] ← "L";
  xord['M'] ← "M"; xord['N'] ← "N"; xord['O'] t "O"; xord['P'] t "P"; xord['Q'] t "Q";
  xord['R'] ← "R"; xord['S'] t "S"; xord['T'] t "T"; xord['U'] ← "U"; xord['V'] ← "V";
  xord['W'] ← "W"; xord['X'] ← "X"; xord['Y'] ← "Y"; xord['Z'] t "Z"; xord['['] t "[";
  xord['\'] ← "\"; xord[']'] ← "]"; xord['^'] t "^"; xord['_'] t "_"; xord['`'] t "`";
  xord['a'] ← "a"; xord['b'] ← "b"; xord['c'] ← "c"; xord['d'] ← "d"; xord['e'] ← "e";
  xord['f'] t "f"; xord['g'] t "g"; xord['h'] t "h"; xord['i'] ← "i"; xord['j'] t "j";
  xord['k'] ← "k"; xord['l'] t "l"; xord['m'] t "m"; xord['n'] t "n"; xord['o'] t "o";
  xord['p'] ← "p"; xord['q'] t "q"; xord['r'] t "r"; xord['s'] t "s"; xord['t'] t "t";
  xord['u'] ← "u"; xord['v'] ← "v"; xord['w'] ← "w"; xord['x'] ← "x"; xord['y'] t "y";
  xord['z'] t "z"; xord['{'] t "{"; xord['|'] t "|"; xord['}'] ← "}"; xord['~'] ← "~";
```

21. In order to help catch errors of badly nested parentheses, PLtoTF assumes that the user will begin each line with a number of blank spaces equal to some constant times the number of open parentheses at the beginning of that line. However, the program doesn't know in advance what the constant is, nor does it want to print an error message on every line for a user who has followed no consistent pattern of indentation.

Therefore the following strategy is adopted: If the user has been consistent with indentation for ten or more lines, an indentation error will be reported. The constant of indentation is reset on every line that should have nonzero indentation.

```
⟨ Globals in the outer block 5) +≡
line: integer;  { the number of the current line }
good-indent: integer;  { the number of lines since the last bad indentation }
indent: integer;  { the number of spaces per open parenthesis, zero if unknown }
level: integer;  { the current number of open parentheses }
```

22. ⟨ Set initial values **6)** +≡

```
line ← 0; good-indent ← 0; indent ← 0; level ← 0;
```

23. The input need not really be broken into lines of any maximum length, and we could read it character by character without any buffering. But we shall place it into a small buffer so that offending lines can be displayed in error messages.

```
⟨ Globals in the outer block 5 ⟩ +≡
left_ln, right-Zn: boolean;  { are the left and right ends of the buffer at end-of-line marks? }
limit: 0 .. buf_size;  { position of the last character present in the buffer}
Zoc: 0 .. buf-size;  { position of the last character read in the buffer }
buffer : array [1 .. buf_size] of char;
input-has-ended: boolean!;  { there is no more input to read }
```

24. ⟨ Set initial values **6)** +≡

```
limit ← 0; Zoc ← 0; left_ln ← true; right-Zn ← true; input-has-ended ← false;
```

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 }
```

26. ⟨Set initial values **6)** +≡

```
chars-on-line ← 0;
```

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 suppies its own.

```
define err-print (#) ≡
        begin if chars-on-line > 0 then print-Zn(' ');
        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-Zn(' (line ', line: 1, '):');
  if ¬left_ln then print('...');
  for k ← 1 to Zoc do print (buffer[k]);  { print the characters already scanned }
  print-Zn (' ');
  if ¬left_ln then print ('   ');
  for k ← 1 to Zoc do print (' ');  { space out the second line }
  for k ← Zoc + 1 to limit do print (buffer [k]);  { print the characters yet unseen }
  if right-Zn then print_ln (' ') else print-Zn('...');
  chars-on-line ← 0;
  end;
```

28. Here is a procedure that does the right thing when we are done reading the present contents of the buffer. It keeps *buffer* [*buf_size*] empty, in order to avoid range errors on certain Pascal compilers.

An infinite sequence of right parentheses is placed at the end of the file, so that the program is sure to get out of whatever level of nesting it is in.

On some systems it is desirable to modify this code so that tab marks in the buffer are replaced by blank spaces. (Simply setting *xord [chr('11')]* ← "␣" would not work; for example, two-line error messages would not come out properly aligned.)

```
procedure fill_buffer;
  begin left_ln ← right-Zn; limit ← 0; Zoc ← 0;
  if left_ln then
    begin if line > 0 then read_ln(pl_file);
    incr (line);
    end;
  if eof (pl_file) then
    begin limit ← 1; buffer [1] ← ')'; right-Zn ← false; input-has-ended ← true;
    end
  else begin while (limit < buf_size − 1) ∧ (¬eoln(pl_file)) do
      begin incr (limit); read (pl_file, buffer [limit]);
      end;
    buffer[limit + 1) ← ' '; right-Zn ← eoln(pl_file);
    if left_ln then ( Set Zoc to the number of leading blanks in the buffer, and check the indentation 29 );
    end;
  end:
```

29. The interesting part about *fill_buffer* is the part that learns what indentation conventions the user is following, if any.

```
define bad-indent (#) ≡
        begin if good-indent ≥ 10 then err-print (#);
        good-indent ← 0; indent ← 0;
        end
```

(Set *loc* to the number of leading blanks in the buffer, and check the indentation 29 ⟩ ≡

```
begin while (Zoc < limit) ∧ (buffer[loc + 1] = ´␣´) do incr( Zoc);
if Zoc < limit then
  begin if level = 0 then
    if Zoc = 0 then incr (good-indent)
    else bad-indent (´Warning : ␣Indented␣line␣occurred␣at␣level␣zero´)
  else if indent = 0 then
      if (Zoc div level) * level = Zoc then
        begin indent + Zoc div level; good-indent ← 1;
        end
      else good-indent ← 0
    else if indent * level = Zoc then incr (good-indent)
      else bad-indent ( ´Warning : ␣Inconsistent␣indentation;␣´,
            ´you␣are␣at␣parenthesis␣level␣´, level : 1);
  end;
end
```

This code is used in section 28.

30. Basic scanning routines. The-global variable ***cur-char*** holds the ASCII code corresponding to the character most recently read from the input buffer, or to a character that has been substituted for the real one.

```
( Globals in the outer block 5) +≡
cur-char : ASCII_code;  { we have just read this }
```

31. Here is a procedure that sets ***cur-char*** to an ASCII code for the next character of input, if that character is a letter or digit. Otherwise it sets ***cur-char*** ← "␣", and the input system will be poised to reread the character that was rejected, whether or not it was a space. Lower case letters are converted to upper case.

```
procedure get-letter-or-digit;
  begin while (Zoc = Zzmit) A (¬right_ln) do fill-buffer;
  if Zoc = limit then cur-char ← "␣"  { end-of-line counts as a delimiter }
  else begin cur-char + xord[buffer[Zoc + 1]];
    if cur-char ≥ "a" then cur-char ← cur_char − '40;
    if ((cur-char ≥"0") A (cur_char ≤"9")) V ((cur-char ≥ "A") A (cur-char ≤ "Z")) then incr(Zoc)
    else cur-char ← "␣";
    end;
  end:
```

32. The following procedure sets ***cur-char*** to the next character code, and converts lower case to upper case. If the character is a left or right parenthesis, it will not be "digested"; the character will be read again and again, until the calling routine does something like *'incr(Zoc)'* to get past it. Such special treatment of parentheses insures that the structural information they contain won't be lost in the midst of other error recovery operations.

```
  define backup ≡
         begin if (cur-char > ")") V (cur-char < "(") then decr (Zoc);
         end  { undoes the effect of get-next }
procedure get-next ;  { sets cur-char to next, balks at parentheses }
  begin while Zoc = limit do fill_buffer;
  incr(Zoc): cur_char ←xord[buffer[Zoc]];
  if cur-char ≥ "a" then
    if cur-char ≤ "z" then cur-char ← cur-char − '40   { uppercasify }
    else begin if cur-char = invalid_code then
        begin err-print ('Illegal␣character␣in␣the␣file'); cur-char ← "?";
        end;
      end
  else if (cur_char ≤")") A (cur-char ≥ "(") then decr(loc);
  end:
```

33. The next procedure is used to ignore the text of a comment, or to pass over erroneous material. As such, it has the privilege of passing parentheses. It stops after the first right parenthesis that drops the level below the level in force when the procedure was called.

```
procedure skip-to-end-of-item;
  var l: integer;  { initial value of level }
  begin l ← level;
  while level ≥ l do
    begin while Zoc = limit do fill_buffer;
    incr(Zoc);
    if buffer[loc] = ')' then decr(level)
    else if buffer[loc] = '(' then incr(level);
    end;
  if input-has-ended then err-print ('File␣ended␣unexpectedly: ␣No␣closing␣")"');
  cur-char ← "␣";  { now the right parenthesis has been read and digested }
  end;
```

34. Sometimes we merely want to skip past characters in the input until we reach a left or a right parenthesis. For example, we do this whenever we have finished scanning a property value and we hope that a right parenthesis is next (except for possible blank spaces).

```
  define skip-to-paren ≡
         repeat get_next until (cur-char = "(") ∨ (cur-char = ")")
  define skip-error (#) ≡
         begin err-print (#); skip-to-paren;
         end  { this gets to the right parenthesis if something goes wrong }
  define flush-error (#) ≡
         begin err-print (#); skip-to-end-of-item;
         end  { this gets past the right parenthesis if something goes wrong }
```

35. After a property value has been scanned, we want to move just past the right parenthesis that should come next in the input (except for possible blank spaces).

```
procedure finish-the-property;  { do this when the value has been scanned }
  begin while cur-char = "␣" do get-next;
  if cur-char ≠ ")" then err-print ('Junk␣after␣property␣value␣will␣be␣ignored');
  skip-to-end-of-item;
  end;
```

36. Scanning property names. We have to figure out the meaning of names that appear in the **PL** file, by looking them up in a dictionary of known keywords. Keyword number ***n*** appears in locations ***start*** $[n]$ through ***start*** $[n+1]-1$ of an array called ***dictionary.***

define ***max-name-index*** = 66 { upper bound on the number of keywords }
define ***max-letters*** = 500 { upper bound on the total length of all keywords }

(Globals in the outer block 5 ⟩ +≡
start: **array** [1 . . ***max-name-index]*** **of** ***0 . . max-letters;***
dictionary: **array** **[0 . .** ***max-letters]*** **of** ***ASCII-code;***
start-ptr: ***0 . . max-name-index;*** { the first available place in ***start*** }
dict_ptr: ***0 . . max-letters;*** { the first available place in ***dictionary*** }

37. (Set initial values 6) +≡
start-ptr ← 1; ***start***[1] ← ***0;*** ***dict_ptr*** ← ***0;***

38. When we are looking for a name, we put it into the ***cur-name*** array. When we have found it, the corresponding ***start*** index will go into the global variable ***name-ptr.***

define ***longest-name*** = 20 { length of DEFAULTRULETHICKNESS }

(Globals in the outer block 5 ⟩ +≡
cur-name: **array** [1 . . ***longest-name]*** **of** ***ASCII-code;*** { a name to look up}
name-length: ***0 . . longest-name;*** { its length}
name-ptr: ***0 . . max-name-index*** ; { its ordinal number in the dictionary }

39. A conventional hash table with linear probing (cf. Algorithm 6.4L in The ***Art of*** Computer ***Programming)*** is used for the dictionary operations. If ***hash[h]*** = 0, the table position is empty, otherwise ***hash***$[h]$ points into the ***start*** array.

define ***hash-prime*** = 101 { size of the hash table }

(Globals in the outer block 5 ⟩ +≡
hash: **array** ***[0 . . hash-prime*** − 1] **of** ***0 . . max-name-index;***
cur-hash: ***0 . . hash-prime*** − 1; { current position in the hash table }

40. (Local variables for initialization 19 ⟩ +≡
h: ***0 . . hash-prime*** − 1; { runs through the hash table }

41. (Set initial values 6 ⟩ +≡
for ***h*** ← ***0*** **to** ***hash-prime*** − 1 **do** ***hush [h]*** ← ***0;***

42. Since there is no chance of the hash table overflowing, the procedure is very simple. After ***lookup*** has done its work, *cur_hash* will point to the place where the given name was found, or where it should be inserted.

```
procedure lookup ; { finds cur-name in the dictionary }
  var k: 0 . . longest-name; { index into cur-name }
    j: 0.. max_letters; { index into dictionary }
    not-found: boolean; { clumsy thing necessary to avoid goto statement }
  begin ( Compute the hash code, cur-hash, for cur-name 43);
  not-found ← true;
  while not-found do
    begin if cur-hash = 0 then cur-hash ← hash-prime − 1 else decr( cur-hash);
    if hash[cur_hash] = 0 then not-found ← false
    else begin j + start [hush [cur-hash]];
      if start [hush [ cur-hush] + 1] = j + name-length then
        begin not-found ← false;
        for k ← 1 to name-length do
          if dictionary [j + k − 1] ≠ cur_name[k] then not-found ← true;
        end;
      end;
    end:
  name-ptr ← hash [cur-hush];
  end;
```

43. (Compute the hash code, ***cur-hash,*** for ***cur-nume*** 43 ⟩ ≡

```
  cur-hash ← cur-name [1];
  for k ← 2 to name-length do cur-hash ← (cur_hash + cur-hash + cur_name[k]) mod hash-prime
```

This code is used in section 42.

44. The "meaning" of the keyword that-begins at ***start***$[k]$ in the dictionary is kept in ***equiv [k]. The*** **numeric** ***equiv*** codes are given symbolic meanings by the following definitions.

```
define comment-code = 0
define check-sum-code = 1
define design-size-code = 2
define design-units-code = 3
define coding-scheme-code = 4
define family_code = 5
define face-code = 6
define seven-bit-safe-flag-code = 7
define header-code = 8
define font-dimen-code = 9
define Zig-table-code = 10
define character-code = 11
define parameter-code = 20
define char-info-code = 50
define width = 1
define height = 2
define depth = 3
define italic = 4
define char-wd-code = char-info-code + width
define char-ht-code = char-info-code + height
define char-dp-code = char-info-code + depth
define char-ic-code = char-info-code + italic
define next-larger-code = 55
define vnr-char-code = 56
define label-code = 70
define Zig-code = 71
define krn-code = 72
define stop-code = 73
```

⟨ Globals in the outer block 5 ⟩ +≡

```
equiv : array [0 .. max_name_index] of byte ;
cur-code: byte:   { equivalent most recently found in equiv }
```

45. We have to get the keywords into the hash table and into the dictionary in the first place (sigh). The procedure that does this has the desired ***equiv*** code as a parameter. In order to facilitate WEB macro writing for the initialization, the keyword being initialized is placed into the last positions of ***cur-name,*** instead of the first positions.

```
procedure enter_name(v : byte):  { cur-name goes into the dictionary}
  var k: 0 .. longest-name:
  begin for k ← 1 to name_length do cur_name[k] ← cur-name [k + longest-name − name-length]:
        { now the name has been shifted into the correct position }
  lookup;  { this sets cur-hush to the proper insertion place }
  hush [cur_hash] ← start-ptr; equiv[start_ptr] ← v;
  for k ← 1 to name-length do
    begin dictionary [ dict_ptr] ← cur_name [k]: incr( dict_ptr);
    end:
  incr( start-ptr); start [start-ptr] ← dict_ptr;
  end:
```

46. Here are the macros to load a name of up to 20 letters into the dictionary. For example, the macro *load5* is used for five-letter keywords.

```
define tail (#) ≡ enter-name (#)
define t20 (#) ≡ cur-name [20] ← #; tail
define t19 (#) ≡ cur-name [19] ← #; t20
define t18 (#) ≡ cur_name[18] ← #; t19
define t17 (#) ≡ cur_name[17] ← #; t18
define t16 (#) ≡ cur_name[16] ← #; t17
define t15 (#) ≡ cur_name[15] ← #; t16
define t14 (#) ≡ cur_name[14] ← #; t15
define t13 (#) ≡ cur_name[13] ← #; t14
define t12 (#) ≡ cur_name[12] ← #; t13
define t11 (#) ≡ cur_name[11] ← #; t12
define t10 (#) ≡ cur_name[10] ← #; tll
define t9 (#) ≡ cur-name [9] ← #; t10
define t8 (#) ≡ cur-name [8] ← #; t9
define t7 (#) ≡ cur_name[7] ← #; t8
define t6 (#) ≡ cur-name [6] ← #; t7
define t5 (#) ≡ cur_name[5] ← #; t6
define t4 (#) ≡ cur_name[4] ← #; t5
define t3 (#) ≡ cur_name[3] ← #; t4
define t2 (#) ≡ cur-name [2] ← #; t3
define tl (#) ≡ cur-name [1] ← #; t2
define load3 ≡ name-length ← 3; t18
define load4 ≡ name-length ← 4; t17
define load5 ≡ name-length ← 5; t16
define load6 ≡ name-length ← 6; t15
define load7 ≡ name-length ← 7; t14
define load8 ≡ name-length ← 8; t13
define load9 ≡ name-length ← 9; t12
define load10 ≡ name-length ← 10; tll
define load11 ≡ name-length ← 11; t10
define load12 ≡ name-length ← 12; t9
define load13 ≡ name-length ← 13; t8
define load14 ≡ name-length ← 14; t7
define load15 ≡ name-length ← 15; t6
define load16 ≡ name-length ← 16; t5
define load17 ≡ name-length ← 17; t4
define load18 ≡ name-length ← 18: t3
define load19 ≡ name-length ← 19; t2
define load20 ≡ name-length ← 20; tl
```

47. (Enter all of the names and their equivalents, except the parameter names 47 ⟩ ≡

```
equiv [0] ← comment-code;  { this is used after unknown keywords }
load8("C")("H")("E")("C")("K")("S")("U")("M")(check_sum_code);
load10("D")("E")("S")("I")("G")("N")("S")("I")("Z")("E")(design_size_code);
load11("D")("E")("S")("I")("G")("N")("U")("N")("I")("T")("S")(design_units_code);
load12("C")("O")("D")("I")("N")("G")("S")("C")("H")("E")("M")("E")(coding_scheme_code);
load6("F")("A")("M")("I")("L")("Y")(family_code);
load4("F")("A")("C")("E")(face_code);
load16("S")("E")("V")("E")("N")("B")("I")("T")
    ("S")("A")("F")("E")("F")("L")("A")("G")(seven_bit_safe_flag_code);
load6("H")("E")("A")("D")("E")("R")(header_code);
load9("F")("O")("N")("T")("D")("I")("M")("E")("N")(font_dimen_code);
load8("L")("I")("G")("T")("A")("B")("L")("E")(lig_table_code);
load9("C")("H")("A")("R")("A")("C")("T")("E")("R")(character_code);
load9("P")("A")("R")("A")("M")("E")("T")("E")("R")(parameter_code);
load6("C")("H")("A")("R")("W")("D")(char_wd_code);
load6("C")("H")("A")("R")("H")("T")(char-ht-code);
load6("C")("H")("A")("R")("D")("P")(char_dp_code);
load6("C")("H")("A")("R")("I")("C")(char_ic_code);
load10("N")("E")("X")("T")("L")("A")("R")("G")("E")("R")(next_larger_code);
load7("V")("A")("R")("C")("H")("A")("R")(var_char_code);
load3("T")("O")("P")(var_char_code + 1);
load3("M")("I")("D")(var_char_code + 2);
load3("B")("0")("T")(var-char-code + 3);
load3("R")("E")("P")(var-char-code + 4);
load3("E")("X")("T")(var-char-code + 4);  { compatibility with older PL format }
load7("C")("O")("M")("M")("E")("N")("T")(comment_code);
load5("L")("A")("B")("E")("L")(label_code);
load3("L")("I")("G")(lig_code);
load3("K")("R")("N")(km-code);
load4("S")("T")("O")("P")(stop_code);
```

This code is used in section 134.

48. ⟨ Enter the parameter names 48 ⟩ ≡

```
load5("S")("L")("A")("N")("T")(parameter_code + 1);
load5("S")("P")("A")("C")("E")(parameter_code + 2);
load7("S")("T")("R")("E")("T")("C")("H")(parameter_code + 3);
load6("S")("H")("R")("I")("N")("K")(parameter_code + 4);
load7("X")("H")("E")("I")("G")("H")("T")(parameter_code + 5);
load4("Q")("U")("A")("D")(parameter_code + 6);
load10("E")("X")("T")("R")("A")("S")("P")("A")("C")("E")(parameter_code + 7);
load4("N")("U")("M")("1")(parameter_code + 8);
load4("N")("U")("M")("2")(parameter_code + 9);
load4("N")("U")("M")("3")(parameter_code + 10);
load6("D")("E")("N")("O")("M")("1")(parameter_code + 11);
load6("D")("E")("N")("O")("M")("2")(parameter_code + 12);
load4("S")("U")("P")("1")(parameter_code + 13);
load4("S")("U")("P")("2")(parameter_code + 14);
load4("S")("U")("P")("3")(parameter_code + 15);
load4("S")("U")("B")("1")(parameter_code + 16);
load4("S")("U")("B")("2")(parameter_code + 17);
load7("S")("U")("P")("D")("R")("O")("P")(parameter_code + 18);
load7("S")("U")("B")("D")("R")("O")("P")(parameter_code + 19);
load6("D")("E")("L")("I")("M")("1")(parameter_code + 20);
load6("D")("E")("L")("I")("M")("2")(parameter_code + 21);
load10("A")("X")("I")("S")("H")("E")("I")("G")("H")("T")(parameter_code + 22);
load20("D")("E")("F")("A")("U")("L")("T")("R")("U")("L")("E")
    ("T")("H")("I")("C")("K")("N")("E")("S")("S")(parameter_code + 8);
load13("B")("I")("G")("O")("P")("S")("P")("A")("C")("I")("N")("G")("1")(parameter_code + 9);
load13("B")("I")("G")("O")("P")("S")("P")("A")("C")("I")("N")("G")("2")(parameter_code + 10);
load13("B")("I")("G")("O")("P")("S")("P")("A")("C")("I")("N")("G")("3")(parameter_code + 11);
load13("B")("I")("G")("O")("P")("S")("P")("A")("C")("I")("N")("G")("4")(parameter_code + 12);
load13("B")("I")("G")("O")("P")("S")("P")("A")("C")("I")("N")("G")("5")(parameter_code + 13);
```

This code is used in section 134.

49. When a left, parenthesis has been scanned, the following routine is used to interpret the keyword that follows. and to store the equivalent value in *cur_code*.

```
procedure get-name:
  begin incr(loc); incr(level): { pass the left parenthesis }
  cur_char ← "␣":
  while cur-char = "␣" do get-next:
  if (cur_char > ")") ∨ (cur-char < "(") then decr(loc): { back up one character }
  name-length ← 0: get-letter-or-digit: { prepare to scan the name }
  while cur_char ≠ "␣" do
    begin if name-length = longest_name then cur_name[1] ← "X" { force error }
    else incr(name_length);
    cur-name[name-length] ← cur_char; get-letter-or-digit:
    end:
  lookup:
  if name_ptr = 0 then err-print('Sorry, ␣I␣don't␣know␣that␣property␣name'):
  cur_code ← equiv[name_ptr];
  end:
```

50. Scanning numeric data. The next thing we need is a trio of subroutines to read the one-byte, four-byte, and real numbers that may appear as property values. These subroutines are careful to stick to numbers between -2^{31} and $2^{31}-1$, inclusive, so that a computer with two's complement 32-bit arithmetic will not be interrupted by overflow.

51. The first number scanner, which returns a one-byte value, surely has no problems of arithmetic overflow.

```
function get-byte: byte;  { scans a one-byte property value }
  var acc: integer;  { an accumulator }
    t: ASCII-code;  { the type of value to be scanned }
  begin repeat get-next;
  until cur-char ≠ "␣";  { skip the blanks before the type code }
  t ← cur-char; acc ← 0;
  repeat get-next;
  until cur-char ≠ "␣";  { skip the blanks after the type code }
  if t = "C" then ⟨ Scan an ASCII character code 52 ⟩
  else if t = "D" then ⟨ Scan a small decimal number 53 ⟩
    else if t = "O" then ⟨ Scan a small octal number 54 ⟩
      else if t = "H" then ⟨ Scan a small hexadecimal number 55 ⟩
        else if t = "F" then ⟨ Scan a face code 56 ⟩
          else skip_error('You␣need␣"C"␣or␣"D"␣or␣"O"␣or␣"H"␣or␣"F"␣here');
  cur-char ← "␣"; get-byte ← acc;
  end;
```

52. The *get-next* routine converts lower case to upper case, but it, leaves the character in the buffer, so we can unconvert it.

```
⟨ Scan an ASCII character code 52 ⟩ ≡
  if (cur-char ≥ '41) A (cur-char ≤ '176) A ((cur-char < "(") ∨ (cur-char > ")")) then
    acc ← xord[buffer[loc]]
  else skip-error('"C"␣value␣must␣be␣standard␣ASCII␣and␣not␣a␣paren')
```

This code is used in section 51.

53. ⟨ Scan a small decimal number 53 ⟩ ≡

```
  begin while (cur-char ≥ "0") A (cur-char ≤ "9") do
    begin acc ← acc * 10 + cur-char − "0";
    if acc > 255 then
      begin skip_error('This␣value␣shouldn't␣exceed␣255'); acc ← 0; cur-char ← "␣";
      end
    else get-next;
    end;
  backup;
  end
```

This code is used in section 51

54. ⟨ Scan a small octal number 54 ⟩ ≡

```
begin while (cur-char ≥ "0") ∧ (cur-char ≤ "7") do
  begin acc ← acc * 8 + cur-char − "0";
  if acc > 255 then
    begin skip_error('This␣value␣shouldn''t␣exceed␣''377'); acc ← 0; cur-char ← "␣";
    end
  else get-next;
  end;
backup;
end
```

This code is used in section 51.

55. ⟨ Scan a small hexadecimal number 55 ⟩ ≡

```
begin while ((cur-char ≥ "0") ∧ (cur-char ≤ "9")) ∨ ((cur-char ≥ "A") ∧ (cur-char ≤ "F")) do
  begin if cur-char ≥ "A" then cur-char ← cur-char + "0" + 10 − "A";
  acc ← acc * 16 + cur-char − "0";
  if acc > 255 then
    begin skip_error('This␣value␣shouldn''t␣exceed␣"FF'); acc ← 0; cur-char ← "␣";
    end
  else get-next;
  end;
backup;
end
```

This code is used in section 51.

56. ⟨ Scan a face code 56 ⟩ ≡

```
begin if cur-char = "B" then acc ← 2
else if cur-char = "L" then acc ← 4
  else if cur-char ≠ "M" then acc ← 18;
get-next;
if cur-char = "I" then incr(acc)
else if cur-char ≠ "R" then acc ← 18;
get-next;
if cur-char = "C" then acc ← acc + 6
else if cur-char = "E" then acc ← acc + 12
  else if cur-char ≠ "R" then acc ← 18:
if acc ≥ 18 then
  begin skip_error('Illegal␣face␣code,␣I␣changed␣it␣to␣MRR'); acc ← 0;
  end;
end
```

This code is used in section 51.

57. The routine that scans a four-byte value puts its output into *cur-bytes,* which is a record containing (yes, you guessed it) four bytes.

⟨ Types in the outer block 17 ⟩ +≡

```
four-bytes = record b0: byte; b1: byte; b2: byte; b3: byte;
  end;
```

58. **define** *c0* ≡ ***cur-bytes.*** *b0*
define ***cl*** ≡ ***cur-bytes.*** *b1*
define *c2* ≡ ***cur-bytes.*** *b2*
define *c3* ≡ ***cur-bytes.*** ***b3***

(Globals in the outer block 5) +≡
cur-bytes: four-bytes; { a four-byte accumulator}

59. Since the ***get-four-bytes*** routine is used very infrequently, no attempt has been made to make it fast; we only want it to work.

```
procedure get-four-bytes; { scans an octal constant and sets four-bytes }
  var c: integer; { leading byte }
    r: integer; { radix }
    q: integer; { 256/r }
  begin repeat get-next;
  until cur-char ≠ "␣"; { skip the blanks before the type code }
  r ← 0; c0 ← 0; c1 ← 0; c2 ← 0; c3 ← 0; { start with the accumulator zero }
  if cur-char = "H" then r ← 16
  else if cur-char = "O" then r ← 8
    else skip-error('An␣octal,␣("O")␣or␣hex␣("H")␣value␣is␣needed␣here');
  if r > 0 then
    begin q ← 256 div r;
    repeat get-next;
    until cur-char ≠ "␣"; { skip the blanks after the type code }
    while ((cur-char ≥ "0") A (cur-char ≤ "9")) ∨ ((cur-char ≥ "A") A (cur-char ≤ "F")) do
      ( Multiply by r, add cur-char − "O", and get-next 60);
    end;
  end;
```

60. (Multiply by ***r,*** add ***cur-char*** − "O", and ***get-next*** 60) ≡

```
  begin if cur-char ≥ "A" then cur-char ← cur-char + "O" + 10 − "A";
  c ← (r * c0) + (c1 divq):
  if c > 255 then
    begin c0 ← 0; c1 ← 0; c2 ← 0; c3 ← 0;
    if r = 8 then skip_error('Sorry,␣the␣maximum␣octal␣value␣is␣0␣37777777777')
    else skip-error ('Sorry,␣the␣maximum␣hex␣value␣is␣H␣FFFFFFFF');
    end
  else if cur-chur ≥ "O" + r then skip-error('Illegal␣digit')
    else begin c0 ← c; c1 ← (r * (c1 modq)) + (c2 divq); c2 ← (r * (c2 modq)) + (c3 divq);
      c3 ← (r * (c3 mod q)) + cur-char − "O"; get-next;
      end;
  end;
```

This code is used in section 59.

61. The remaining scanning routine is the most interesting. It scans a real constant and returns the nearest *fix_word* approximation to that constant. **A *fix-word*** is a 32-bit integer that represents a real value that has been multiplied by 2^{20}. Since PLtoTF restricts the magnitude of reals to 1024, the ***fix-word*** will have a magnitude less than 2^{30}.

define ***unity*** ≡ '4000000 { 2^{20}, the ***fix-word*** 1.0 }

(Types in the outer block 17) +≡
fix-word = ***integer;*** { a scaled real value with 20 bits of fraction }

62. When a real value is desired, we might as well treat 'D' and 'R' formats as if they were identical.

```
function get-fix : fix-word :  { scans a real property value }
  var negative: boolean;  { was there a minus sign? }
    ucc: integer;  { an accumulator }
    int-purt: integer;  {the integer part }
    j: 0 . . 7;  { the number of decimal places stored }
  begin repeat get-next;
  until cur-char ≠ "␣";  { skip the blanks before the type code }
  negative ← false; ucc ← 0;  { start with the accumulators zero }
  if (cur-char ≠ "R") A (cur-char ≠ "D") then skip-error( 'An␣"R"␣or␣"D"␣value␣is␣needed␣here')
  else begin ( Scan the blanks and/or signs after the type code 63);
    while (cur-chur ≥ "0") A (cur-chur ≤ "9") do ( Multiply by 10, add cur-char − "0", and get-next 64 );
    int-purt ← acc; ucc ← 0;
    if cur-char = "." then ( Scan the fraction part and put it in ucc 66 );
    if (acc ≥ unity) A (int_part = 1023) then skip_error( 'Real␣constants␣must␣be␣less␣than␣1024')
    else ucc ← int-purt * unity + ucc:
    end:
  if negativc then get-fix ← −acc else get-fix ← ucc;
  end:
```

63. (Scan the blanks and/or signs after the type code 63) ≡

```
  repeat get-next;
    if cur-char = "-" then
      begin cur-chur ← "␣"; negative ← true:
      end
    else if cur_char = "+" then cur-char ← "␣";
  until cur-char ≠ "␣"
```

This code is used in section 62.

64. (Multiply by 10, add **cur-char** − "0", and **get-next** 64) ≡

```
  begin ucc ← ucc * 10 + cur-chur − "0";
  if ucc ≥ 1024 then
    begin skip_error( 'Real␣constants␣must␣be␣less␣than␣1024'); acc ← 0; cur-char ← "␣";
    end
  else get-next;
  end
```

This code is used in section 62.

65. To scan the fraction $.d_1 d_2 \ldots$, we keep track of up to seven of the digits d_j. **A** correct result is obtained if we first compute $f' = \lfloor 2^{21}(d_1 \ldots d_j)/10^j \rfloor$, after which $f = \lfloor (f' + 1)/2 \rfloor$. It is possible to have $f = 1.0$.

(Globals in the outer block 5) +≡

```
fraction-digits: array [1 . . 7] of integer:  { 2^21 times d_j }
```

66. (Scan the fraction part and put it in ucc 66) ≡

```
begin j ← 0; get-next;
while (cur-char ≥ "0") ∧ (cur-char ≤ "9") do
  begin if j < 7 then
    begin incr (j); fraction-digits [j] ← 10000000 * (cur-char − "0");
    end;
  get-next;
  end;
ucc ← 0;
while j > 0 do
  begin ucc ← fraction_digits[j] + (acc div 10); decr(j);
  end;
ucc ← (ucc + 10) div 20;
end
```

This code is used in section 62.

67. Storing the property values. When property values have been found, they are squirreled away in a bunch of arrays. The header information is unpacked into bytes in an array called ***header-bytes.*** The ligature/kerning program is stored in an array of type ***four-bytes;*** note that such a program is at most 511 steps long, since a label after step 255 may begin a program of length 256. Another ***four-bytes*** array holds the specifications of extensible characters. The kerns and parameters are stored in separate arrays of ***fix-word*** values.

Instead of storing the design size in the header array, we will keep it in a ***fix-word*** variable until the last minute. The number of units in the design size is also kept in a ***fix-word.***

```
⟨ Globals in the outer block 5 ⟩ +≡
header-bytes: array [header-index] of byte; {the header block}
header-ptr : header-index ; { the number of header bytes in use }
design-size : fix-word ; { the design size }
design-units : fix-word; { reciprocal of the scaling factor }
seven_bit_safe_flag: boolean; { does the file claim to be seven-bit-safe? }
Zig-kern: array [0 . . 511] of four-bytes; {the ligature program}
nl: 0 . . 511; { the number of ligature/kern instructions so far}
unused-label: boolean; { was the last lig/kern step a label? }
kern: array [0 . . 256] of fix-word; { the distinct kerning amounts }
nk: 0 . . 256; { the number of entries of kern }
exten: array [0 . . 255] of four-bytes; { extensible character specs }
ne: 0 . . 256; { the number of extensible characters }
param: array [1 . . max_param_words] of fix-word; { f ontdimen parameters }
np: 0.. max_param_words: { the largest parameter set nonzero }
check-sum-specified: boolean: { did the user name the check sum? }
```

68. (Types in the outer block 17) +≡

```
header-index = 0 . . mux-header-bytes ;
```

69. (Local variables for initialization 19) +≡

```
d: header-index; { an index into header-bytes }
```

70. We start by setting up the default values.

```
define check-sum-Zoc = 0
define design-size-Zoc = 4
define coding-scheme-Zoc = 8
define family_loc = coding-scheme-Zoc + 40
define seven_flag_loc = family_loc + 20
define face_loc = seven_flag_loc + 3
```

(Set initial values 6) 4-E

```
check-sum-specified ← false;
for d ← 0 to 18 * 4 − 1 do header_bytes[d] ← 0;
header-bytes [8] t 11; header_bytes[9] ← "U"; header_bytes[10] ← "N"; header_bytes[11] ← "S";
header_bytes[12] ← "P"; header_bytes[13] t "E"; header_bytes[14] ← "C"; header_bytes[15] ← "I";
header_bytes[16] ← "F"; header_bytes[17] ← "I"; header_bytes[18] ← "E"; header_bytes[19] ← "D";
for d ← family_loc to family_loc + 11 do header-bytes[d] ← header-bytes[d − 40];
design-size ← 10 * unity; design-units ← unity; seven-bit-safe-flag ← false;
header-ptr ← 18 * 4; nl ← 0; unused-label ← false; nk ← 0; ne ← 0; np ← 0;
```

71. Most of the dimensions, however, go into the ***memory*** array. There are at most 257 widths, 257 heights, 257 depths, and 257 italic corrections, since the value 0 is required but it need not be used. So ***memory*** has room for 1028 entries, each of which is a ***fix-word.*** An auxiliary table called ***link*** is used to link these words together in linear lists, so that sorting and other operations can be done conveniently.

We also add four "list head" words to the ***memory*** and ***link*** arrays; these are in locations ***width*** through ***italic,*** i.e., 1 through 4. For example, *link*[*height*] points to the smallest element in the sorted list of distinct heights that have appeared so far, and ***memory[height]*** is the number of distinct heights.

```
define mem-size = 1028 + 4  { number of nonzero memory addresses }
```

(Types in the outer block 17 ⟩ +≡

```
pointer = 0 . . mem-size;  { an index into memory }
```

72. The arrays ***char-wd, char-ht, char-dp,*** and ***char-ic*** contain pointers to the ***memory*** array entries where the corresponding dimensions appear. Two other arrays, ***char-tag*** and ***char-remainder,*** hold the other information that TFM files pack into a ***char-info-word.***

```
define no-tag = 0  { vanilla character }
define Zig-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 }
```

(Globals in the outer block 5) +≡

```
memory: array [pointer] of fix-word;  { character dimensions and kerns }
mem-ptr : pointer;  { largest memory word in use }
link: array [pointer] of pointer;  { to make lists of memory items }
char-wd: array [byte] of pointer;  { pointers to the widths }
char-ht: array [byte] of pointer;  { pointers to the heights }
char-dp: array [byte] of pointer;  { pointers to the depths }
char-ic: array [byte] of pointer;  { pointers to italic corrections }
char-tag: array [byte] of no-tag . . ext-tag;  {character tags}
char-remainder: array [byte] of 0 . . 255;
        { pointers to ligature labels, next larger characters, or extensible characters}
```

73. (Local variables for initialization 19 ⟩ +≡

```
c: byte;  { runs through all character codes }
```

74. (Set initial values 6 ⟩ +≡

```
for c ← 0 to 255 do
  begin char-wd [c] ← 0; char-ht [c] ← 0; char-dp[c] ← 0; char-ic[c] + 0;
  char-tag[c] ← no-tag; char-remainder[c] ← 0;
  end;
memory [0] ← '17777777777;  { an "infinite" element at the end of the lists }
memory [width] ← 0; link [width] ← 0;  { width list is empty }
memory [height] ← 0; link [height] ← 0;  { height list is empty }
memory [depth] ← 0; link [depth] ← 0;  { depth list is empty }
memory [italic] ← 0; link [italic] ← 0;  { italic list is empty }
mem-ptr + italic;
```

75. As an example of these data structures, let us consider the simple routine that inserts a potentially new element into one of the dimension lists. The first, parameter indicates the list head (i.e., ***h = width*** for the width list, etc.); the second parameter is the value that is to be inserted into the list if it is not already present. The procedure returns the value of the location where the dimension appears in ***memory.*** The fact that ***memory [0]*** is larger than any legal dimension makes the algorithm particularly short.

We do have to handle two somewhat subtle situations. A width of zero must be put into the list, so that a zero-width character in the font will not appear to be nonexistent (i.e., so that its ***char-wd*** index will not be zero), but this does not need to be done for heights, depths, or italic corrections. Furthermore, it is necessary to test for memory overflow even though we have provided room for the maximum number of different dimensions in any legal font, since the PL file might foolishly give any number of different sizes to the same character.

```
function sort-in(h : pointer; d : fix-word): pointer;  { inserts into list }
  var p: pointer;  { the current node of interest }
  begin if (d = 0) A (h ≠ width) then sort-in ← 0
  else begin p ← h;
    while d ≥ memory [link [p]] do p ← link [p];
    if (d = memory [p]) A (p ≠ h) then sort-in ← p
    else if mem-ptr = mem_size then
        begin err-print ('Memory␣overf low :␣more␣than␣1028␣widths,␣etc');
        print_ln( 'Congratulations ! ␣It''s␣hard␣to␣make␣this␣error .'); sort-in ← p;
        end
      else begin incr( mem-ptr): memory [mem-ptr] ← d; Zink[mem-ptr] ← link[p]; link[p] ← mem-ptr;
        incr (memory [h]); sort-in ← mem-ptr;
        end:
    end:
  end;
```

76. When these lists of dimensions are eventually written to the TFM file, we may have to do some rounding of values, because the TFM file allows at most 256 widths, 16 heights, 16 depths, and 64 italic corrections. The following procedure takes a given list head ***h*** and a given dimension ***d,*** and returns the minimum ***m such*** that the elements of the list can be covered by m intervals of width ***d.*** It also sets ***next-d*** to the smallest value ***d'*** > ***d*** such that the covering found by this procedure would be different. In particular, if ***d*** = 0 it computes the number of elements of the list. and sets ***next-d*** to the smallest distance between two list elements. (The covering by intervals of width ***next-d*** is not guaranteed to have fewer than m elements, but in practice this seems to happen most of the time.)

```
⟨ Globals in the outer block 5 ⟩ +≡
next-d: fix-word;  { the next larger interval that is worth trying }
```

77. Once again we can make good use -of the fact that **memory** [0] is "infinite."

```
function min_cover(h : pointer; d : fix-word): integer;
  var p: pointer;  { the current node of interest }
    l: fix-word;  { the last element covered by the current interval }
    m: integer;  { the current size of the cover being generated }
  begin m ← 0; p ← link[h]; next-d ← memory[O];
  while p ≠ 0 do
    begin incr (m); l ← memory [p];
    while memory [link [p]] ≤ l + d do p ← link [p];
    p ← link[p];
    if memory [p] − l < next-d then next-d ← memory [p] − l;
    end;
  min_cover ← m;
  end:
```

78. The following procedure uses *min_cover* to determine the smallest **d** such that a given list can be covered with at most a given number of intervals.

```
function shorten(h : pointer; m : integer): fix-word;  { finds best way to round }
  var d: fix-word;  { the current trial interval length }
    k: integer;  { the size of a minimum cover }
  begin if memory [h] > m then
    begin k ← min_cover(h, 0); d ← next-d;  { now the answer is at least d }
    repeat d ← d + d; k ← min_cover(h, d);
    until k ≤ m;  { first we ascend rapidly until finding the range }
    d ← d div 2: k ← min_cover(h, d):  { now we run through the feasible steps }
    while k > m do
      begin d ← next-d: k ← min_cover(h. d);
      end;
    shorten ← d;
    end
  else shorten ← 0:
  end:
```

79. When we are nearly ready to output the TFM file. we will set **index** [p] ← **k** if the dimension in **memory** [p] is being rounded to the kth element of its list.

```
( Globals in the outer block 5) +≡
index: array [pointer] of byte;
```

80. Here is the procedure that sets the ***index*** values. It also shortens the list so that there is only one element per covering interval; the remaining elements are the midpoints of their clusters.

```
procedure set-indices(h : pointer; d : fix_word); { reduces and indexes a list }
  var p: pointer; { the current node of interest }
    q: pointer; { trails one step behind p }
    m: byte; { index number of nodes in the current interval }
    l: fix_word; { 1eastva1ue in the current interval }
  begin q ← h; p ← link[q]; m ← 0;
  while p ≠ 0 do
    begin incr(m); l ← memory [p]; index [p] ← m;
    while memory [link [p]] ≤ l + d do
      begin p ← link [p]; index[p] ← m;
      end;
    link [q] ← p; memory [p] ← (l + memory [p]) div 2; q ← p; p ← link [p];
    end;
  memory [h] ← m;
  end;
```

81. The input phase. We're ready now to read and parse the PL file, storing property values as we go.

```
⟨ Globals in the outer block 5⟩ +≡
c: byte; { the current character or byte being processed }
```

82. ⟨ Read all the input 82⟩ ≡

```
cur-char ← " ";
repeat while cur-char = " " do get-next;
  if cur-char = "(" then ⟨ Read a font property value 84 ⟩
  else if (cur-char = ")") ∧ ¬input_has_ended then
      begin err-print ( ´Extra right parenthesis´); incr (Zoc); cur-char ← " ";
      end
    else if -input-has-ended then junk-error;
until input-has-ended
```

This code is used in section 134.

83. The *junk-error* routine just referred to is called when something appears in the forbidden area between properties of a property list.

```
procedure junk-error; { gets past no man's land }
  begin err-print ( There´´s junk here that is not in parentheses´); skip-to-paren;
  end;
```

84. For each font property, we are supposed to read the data from the left parenthesis that is the current value of *cur-char* to the right parenthesis that matches it in the input. The main complication is to recover with reasonable grace from various error conditions that might arise.

```
⟨ Read a font property value 84 ⟩ ≡
  begin get-name;
  if cur-code = comment-code then skip-to-end-of-item
  else if cur-code > character-code then
      flush_error(´This property name doesn´´t belong on the outer level´)
    else begin ⟨ Read the font property value specified by cur-code 85⟩;
      finish-the-property;
      end;
  end
```

This code is used in section 82.

85. ⟨ Read the font property value specified by *cur-code* 85⟩ ≡

```
case cur-code of
check-sum-code: begin check-sum-specified ← true; read-four-bytes ( check-sum-Zoc);
  end;
design-size-code: ⟨ Read the design size 88 ⟩;
design-units-code: ⟨Read the design units 89⟩;
coding-scheme-code: read-BCPL( coding-scheme-Zoc, 40);
family-code: read-BCPL(famiZy-Zoc, 20);
face-code: header-bytespace-Zoc] ← get-byte;
seven-bit-safe-flag-code: ⟨ Read the seven-bit-safe flag 90 ⟩;
header-code: ⟨ Read an indexed header word 91 ⟩;
font-dimen-code: ⟨ Read font parameter list 92 ⟩;
Zig-table-code: read-Zig-kern;
character-code: read-char-info;
end
```

This code is used in section 84.

86. The **case** statement just given makes use of two subroutines that we haven't defined yet. The first of these puts a 32-bit octal quantity into four specified bytes of the header block.

```
procedure read_four_bytes(l : header-index);
  begin get-four-bytes; header_bytes[l] ← c0; header_bytes[l + 1] ← c1; header_bytes[l + 2] ← c2;
  header-bytes [Z + 3] ← c3;
  end;
```

87. The second little procedure is used to scan a string and to store it in the "BCPL format" required by TFM files. The string is supposed to contain at most **n** bytes, including the first byte (which holds the length of the rest of the string).

```
procedure read_BCPL(l : header-index; n : byte);
  var k: header-index;
  begin k ← l;
  while cur-char = "␣" do get-next;
  while (cur-char ≠ "(") ∧ (cur-char ≠ ")") do
    begin if k < l + n then incr(k);
    if k < l + n then header-bytes[k] ← cur-char;
    get-next;
    end;
  if k = l + 71 then
    begin err-print('String␣is␣too␣long;␣its␣first␣', n − 1 : 1, '␣characters␣will␣be␣kept');
    decr(k);
    end;
  header-bytes[Z] ← k − l;
  while k < l + n − 1 do {tidy up the remaining bytes by setting them to nulls}
    begin incr(k); header-bytes[k] ← 0;
    end;
  end;
```

88. ⟨ Read the design size 88 ⟩ ≡

```
  begin next-d ← get-fix;
  if (next-d < unity) ∨ (next-d ≥ '10000000000) then
    err-print('The␣design␣size␣must␣be␣between␣1␣and␣1024')
  else design-size ← next-d;
  end
```

This code is used in section 85.

89. ⟨ Read the design units 89 ⟩ ≡

```
  begin next-d ← get-fix;
  if next-d ≤ 0 then err_print('The␣number␣of␣units␣per␣design␣size␣must␣be␣positive')
  else design-units ← next-d;
  end
```

This code is used in section 85.

90. ⟨ Read the seven-bit-safe flag 90 ⟩ ≡

```
  begin while cur-char = "␣" do get-next;
  if cur-char = "T" then seven_bit_safe_flag ← true
  else if cur-char = "F" then seven_bit_safe_flag ← false
    else err-print('The␣f lag␣value␣should␣be␣"TRUE"␣or␣"FALSE"');
  skip-to-paren;
  end
```

This code is used in section 85.

91. ⟨ Read an indexed header word 91 ⟩ ≡

```
begin c ← get-byte;
if c < 18then skip_error('HEADER␣indices␣should␣be␣18␣or␣more')
else if 4 * c + 4 > max_header_bytes then
    skip_error('This␣HEADER␣index␣is␣too␣big␣for␣my␣present␣table␣size')
  else begin while header-ptr < 4 * c do
      begin header-bytes [ header-ptr] ← 0; incr (header-ptr );
      end;
    read-four-bytes(4 * c); header-ptr ← 4 * c + 4;
    end;
end
```

This code **is** used **in section 85.**

92. The remaining kinds of font property values that need to be read are those that involve property lists on higher levels. Each of these has a loop similar to the one that was used at level zero. Then we put the right parenthesis back so that '*finish_the_property*' will be happy; there is probably a more elegant way to do this.

```
define finish_inner_property_list ≡
        begin decr (Zoc); incr (level); cur-char ← " ) ";
        end
```

⟨ Read font parameter list 92 ⟩ ≡

```
begin while level = 1 do
  begin while cur-char = "␣" do get-next;
  if cur-char = " (" then ⟨ Read a parameter value 93 ⟩
  else if cur_char = ") " then skip-to-end-of-item
    else junk-error;
  end;
finish-inner-property-Z&;
end
```

This code is used in section 85.

93. ⟨ Read a parameter value 93 ⟩ ≡

```
begin get-name ;
if cur-code = comment-code then skip-to-end-of-item
else if (cur-code < parameter-code) ∨ (cur-code ≥ char_wd_code) then
    flush_error('This␣property␣name␣doesn''t␣belong␣in␣a␣FONTDIMEN␣list')
  else begin if cur-code = parameter-code then c ← get-byte
    else c ← cur-code − parameter-code;
    if c = 0 then flush_error('PARAMETER␣index␣must␣not␣be␣zero')
    else if c > max_param_words then
        flush_error('This␣PARAMETER␣index␣is␣too␣big␣for␣my␣present␣table␣size')
      else begin while np < c do
          begin incr (np); param [np] ← 0;
          end:
        param [c] ← get_fix: finish_the_property:
        end:
    end;
end
```

This code is used in section 92.

94. ⟨ Read ligature/kern list 94 ⟩≡
```
begin while level = 1 do
  begin while cur-char = "␣" do get-next;
  if cur-char = "(" then ⟨ Read a ligature/kern command 95 ⟩
  else if cur-char = ")" then skip-to-end-of-item
    else junk-error;
  end;
finish-inner-property-list;
end
```
This code is used in section 134.

95. ⟨ Read a ligature/kern command 95 ⟩≡
```
begin get-name;
if cur-code = comment-code then skip-to-end-of-item
else if (cur-code < label-code) ∨ (cur-code > stop-code) then
    flush_error('This␣property␣name␣doesn''t␣belong␣in␣a␣LIGTABLE␣list')
  else begin case cur-code of
    label_code: ⟨ Read a label step 97 ⟩;
    Zig-code: ⟨ Read a ligature step 98 ⟩;
    km-code: ⟨ Read a kerning step 100 ⟩;
    stop-code: ⟨ Read a stop step 101 ⟩;
    end;
    finish-the-property;
    end;
end
```
This code is used in section 94.

96. When a character is about to be tagged, we call the following procedure so that an error message is given in case of multiple tags.
```
procedure check_tag(c : byte);  { print error if c already tagged }
  begin case char-tag[c] of
  no-tag: do-nothing;
  Zig-tag: err-print ('This␣character␣already␣appeared␣in␣a␣LIGTABLE␣LABEL');
  list_tag: err-print ('This␣character␣already␣has␣a␣NEXTLARGER␣spec');
  ext_tag: err-print ('This␣character␣already␣has␣a␣VARCHAR␣spec');
  end;
  end;
```

97. ⟨ Read a label step 97 ⟩≡
```
begin c ← get-byte; check-tag(c);
if nl > 255 then
  err_print('LIGTABLE␣with␣more␣than␣255␣commands␣cannot␣have␣further␣labels')
else begin char_tag[c] ← Zig-tag; char-remainder[c] ← nl; unused-label ← true;
  end;
end
```
This code is used in section 95.

98. (Read a ligature step **98)** ≡
```
begin lig_kern[nl].b0 ← 0; lig_kern[nl].b1 ← get-byte; lig_kern[nl].b2 ← 0; lig_kern[nl].b3 ← get-byte;
if nl = 511 then err_print('LIGTABLE should never exceed 511 LIG/KRN commands')
else incr(nl);
unused-label ← false;
end
```
This code is used in section 95.

99. **define** *stop-flag* = 128 { value indicating STOP' in a lig/kern program }
define *kern_flag* = 128 { op code for a kern step }

(Globals in the outer block 5) +≡
```
km-ptr: 0 . . 256; { an index into kern }
```

100. (Read a kerning step 100 ⟩≡
```
begin lig_kern[nl].b0 ← 0; lig_kern[nl].b1 ← get-byte; lig_kern[nl].b2 ← kern_flag; kern[nk] ← get-fix;
km-ptr ← 0;
while kern [krn_ptr] ≠ kern [nk] do incr (km-ptr);
if km-ptr = nk then
  begin if nk < 256 then incr (nk)
  else begin err-print ('At most 256 dif f erent kerns are allowed'); km-ptr ← 255;
    end;
  end;
Zig-kern[nl]. b3 ← krn_ptr ;
if nl = 511 then err-print ('LIGTABLE should never exceed 511 LIG/KRN commands')
else incr(nl);
unused-label ← false;
end
```
This code is used in section 95.

101. (Read a stop step 101 ⟩≡
```
begin if nl = 0 then err-print ('Why STOP? You haven''t started')
else begin if unused-label then
    begin err-print ('STOP after LABEL invalidates the label');
    for c ← 0 to 255 do
      if (char-fag[c] = Zig-tag) A (char_remainder[c] = nl) then char_tag[c] ← no-tag;
    unused-label ← false;
    end:
  lig_kern[nl − 1].b0 ← stop-flag;
  end;
end
```
This code is used in section 95.

102. 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 102 ⟩≡
  begin c ← get-byte;  { read the character code that is being specified }
  (Print c in octal notation 107);
  while level = 1 do
    begin while cur-char = "␣" do get-next;
    if cur-char = "(" then ⟨ Read a character property 103 ⟩
    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 134.

```
103. ⟨ Read a character property 103 ⟩≡
  begin get-name;
  if cur-code = comment-code then skip-to-end-of-item
  else if (cur-code < char-wd-code) ∨ (cur-code > var-char-code) then
      flush_error('This␣property␣name␣doesn''t␣belong␣in␣a␣CHARACTER␣list')
    else begin case cur-code of
      char-wd-code: char-wd [c] ← sort-in (width, get-fix);
      char-ht-code: char-ht [c] ← sort-in( height, get-fix);
      char-dp-code: char-dp [c] ← sort-in ( depth, get_fix);
      char-ic-code: char-ic [c] ← sort-in( italic. get-fix);
      next_larger_code: begin check-tag(c); char_tag[c] ← list_tag; char-remainder[c] ← get-byte;
        end:
      var-char-code: ⟨ Read an extensible recipe for c 104 ⟩;
      end:
      finish-the-property;
      end;
  end
```

This code is used in section 102.

```
104. ⟨ Read an extensiblt recipe for c 104 ⟩≡
  begin if ne = 256 then err-print ('At␣most␣256␣VARCHAR␣specs␣are␣allowed')
  else begin check_tag(c): char_tag[c] ← ext_tag; char-remainder[c] ← ne;
    exten[ne].b0 ← 0: exten[ne].b1 ← 0: exten[ne].b2 ← 0: exten[ne].b3 ← 0:
    while level = 2 do
      begin while cur-char = "␣" do get-next:
      if cur_char = "(" then ⟨ Read an extensible piece 105 ⟩
      else if cur_char = ")" then skip-to-end-of-item
        else junk_error;
      end:
    incr( ne): finish_inner property_list;
    end:
  end
```

This code is used in section 103.

105. ⟨ Read an extensible piece 105 ⟩ ≡

```
begin get-name;
if cur-code = comment-code then skip-to-end-of-item
else if (cur-code < var-char-code + 1) ∨ (cur-code > var-char-code + 4) then
    flush_error('This␣property␣name␣doesn''t␣belong␣in␣a␣VARCHAR␣list')
  else begin case cur-code − (varxhar-code + 1) of
    0: exten[ne].b0 ← get-byte;
    1: exten[ne].b1 ← get-byte;
    2: exten[ne].b2 ← get-byte;
    3: exten[ne].b3 ← get-byte;
    end;
    finish- the-property ;
    end;
end
```

This code is used in section 104.

106. The input routine is now complete except for the following code, which prints a progress report as the file is being read.

```
procedure print-octaZ(c : byte);  { prints three octal digits}
  begin print ('''', (c div 64) : 1, ((c div 8) mod 8) : 1, (c mod 8) : 1);
  end;
```

107. ⟨ Print c in octal notation 107) ≡

```
begin if chars-on-line = 8 then
    begin print-Zn ('␣'); chars-on-line ← 1;
    end
  else begin if chars-on-line > 0 then print ('␣');
    incr (chars-on-line);
    end;
  print-octal(c);  { progress report }
  end
```

This code is used in section 102.

108. The checking and massaging phase. Once the whole PL file has been read in, we must check it for consistency and correct any errors. This process consists mainly of running through the characters that exist and seeing if they refer to characters that don't exist. **We** also compute the true value of ***seven-unsafe;*** we make sure that the charlists contain no loops; and we shorten the lists of widths, heights, depths, and italic corrections, if necessary, to keep from exceeding the required maximum sizes.

```
( Globals in the outer block 5) +≡
seven-unsafe : boolean;  { do seven-bit characters generate eight-bit ones? }
```

109.
```
( Correct and check the information 109 ⟩≡
  ( Make sure the ligature/kerning program ends with STOP' 110);
  seven-unsafe ← false;
  for c ← 0 to 255 do
    if char-wd [c] ≠ 0 then ( For all characters g generated by c, make sure that char-wd [g] is nonzero,
        and set seven-unsafe if c < 128 ≤ g 111 ⟩;
  if seven_bit_safe_flag A seven-unsafe then print-ln ('The␣font␣is␣not␣really␣seven-bit-safe!');
  ( Doublecheck the lig/kern commands and the extensible recipes 115 ⟩;
  for c ← 0 to 255 do ( Make sure that c is not the largest element of a charlist cycle 116);
  ( Put the width, height? depth, and italic lists into final form 118)
```
This code is used in section 134.

110.
```
(Make sure the ligature/kerning program ends with STOP' 110 ⟩≡
  if unused-label then
    begin for c ← 0 to 255 do
      if (char_tag[c] = Zig-tug) A (char_remainder[c] = nl) then char_tag[c] ← no-tug;
    print_ln('Last␣LIGTABLE␣LABEL␣was␣not␣used.');
    end;
  if nl > 0 then lig_kern [nl − 1]. b0 ← stop-flag
```
This code is used in section 109.

111. The checking that we need in several places is accomplished by two macros that are only slightly tricky.

```
define existence-tail (#)≡
        begin char-wd[g] ← sort-in (width, 0); print (#, '␣'); print-octal(c);
        print_ln('␣had␣no␣CHARACTER␣spec.');
        end;
      end
define check-existence (#)≡
      begin g ← #;
      if (g ≥ 128) A (c < 128) then seven-unsafe ← true;
      if char-wd [g] = 0 then existence-tail
```

```
( For all characters g generated by c, make sure that char,wd [g] is nonzero, and set seven-unsafe if
    c< 128 ≤ g 111) ≡
case char_tag[c] of
no-tag: do-nothing;
Zig-tag: (Check ligature program of c 113 ⟩;
list-tag: check_existence(char_remainder [c])('The␣character␣NEXTLARGER␣than');
ext_tag: (Check the pieces of exten[c] 114 ⟩;
end
```
This code is used in section 109.

112. (Globals in the outer block 5) +≡
lig_ptr: 0 . . 511; { an index into *Zig-kern* }

113. (Check ligature program of c 113) ≡
```
begin if char-wd [c] = 0 then
  begin print ( There''s␣a␣LABEL␣but␣no␣CHARACTER␣spec␣f or,); print-octal(c); print-Zn( ' . ');
  char-wd [c] ← sort-in (width, 0);
  end;
Zig-ptr ← char-remainder [c];
repeat if lig_kern [Zig-ptr]. b2 < kern_flag then
    begin check-existence (Zig-kern [Zig-ptr]. b1 )( 'LIG␣character␣generated␣by');
    check_existence(lig_kern[lig_ptr].b3)('LIG␣character␣generated␣by');
    end
  else check_existence(lig_kern[lig_ptr].b1 )( 'KRN␣character␣generated␣by');
  incr (Zig-ptr);
until lig_kern[lig_ptr − 1].b0 = stop_flag;
end
```
This code is used in section 111.

114. (Check the pieces of *exten[c]* 114) ≡
```
begin if exten [char-remainder [c]]. b0 > 0 then
  check-existence (exten [ char-remainder [c]]. bO)('TOP␣piece␣of ␣character');
if exten [ char-remainder [c]]. b1 > 0 then
  check-existence( exten [ char-remainder [c]]. b1 )( 'MID␣piece␣of ␣character');
if exten [ char-remainder [c]]. b2 > 0 then
  check-existence ( exten [char-remainder [c]]. b2 )( 'BOT␣piece␣of ␣character');
check-existence (exten [char-remainder [c]]. b3 )( 'REP␣piece␣of "character');
end
```
This code is used in section 111.

115. The lig/kern program may still contain references to nonexistent characters, if parts of that program are never used. Similarly, there may be extensible characters that are never used, because they were overridden by NEXTLARGER, say. This would produce an invalid TFM file; so we must fix such errors.

```
define double-check-tail (#) ≡
    if char-wd [0] = 0 then char-wd [0] ← sort-in( width, 0);
  print ( 'Unused,', #, ' refers to nonexistent character '); print-octal(c); print-Zn( . ! ');
  end ;
  end
define double-check-Zig (#) ≡
  begin c ← lig_kern[lig_ptr].#;
  if char-wd [c] = 0 then
    begin Zig-kern, [lig_ptr].# ← 0; double_check_tail
define double_check_ext (#) ≡
  begin c ← exten[g].#;
  if c > 0 then
    if char-wd [c] = 0 then
      begin exten[g].# ← 0: double-check-tail
define double-check-rep(t) ≡
  begin c ← exten[g].#;
  if char-wd [c] = 0 then
    begin exten [g] .# ← 0; double-check-tail
```

⟨ Doublecheck the lig/kern commands and the extensible recipes 115 ⟩ ≡

```
if nl > 0 then
  for Zig-ptr ← 0 to nl − 1 do
    if lig_kern[lig_ptr]. b2 < kern_flag then double_check_lig(b3)('LIG step')
    else double-check-Zig( b1)('KRN step');
if ne > 0 then
  for g ← 0 to ne − 1 do
    begin double-check-ext (b0)('VARCHAR TOP'); double-check-ext( b1)('VARCHAR MID');
    double-check-ext (b2)('VARCHAR BOT'); double-check-rep (b3 )('VARCHAR REP');
    end
```

This code is used in section 109.

116. ⟨ Make sure that c is not the largest element of a charlist cycle 116) ≡

```
if char_tag[c] = list_tag then
  begin g ← char-remainder [c];
  while (g < c) A (char_tag[g] = list-tag) do g ← char-remainder [g];
  if g = c then
    begin char_tag[c] ← no-tag:
    print ('A cycle of NEXTLARGER characters has been broken at '); print-octal(c):
    print-Zn ('.');
    end:
  end
```

This code is used in section 109.

117. ⟨ Globals in the outer block 5) +≡

```
delta: fix-word: { size of the intervals needed for rounding }
```

118. **define** *round-message* (#) ≡
  if *delta* > **0** **then**
    print_ln(´I␣had␣to␣round␣some␣´,#,´s␣by´,(((*delta* + 1) **div** 2)/´4000000):1:7,´␣units.´)

(Put the width, height, depth, and italic lists into final form **118**) ≡
  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 109.

119. 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(#) ≡ write(tfm_file,#)
```

120. The general plan for producing TFM files is long but simple:

```
(Do the output 120) ≡
  (Compute the twelve subfile sizes 122);
  (Output the twelve subfile sizes 123);
  (Output the header block 125);
  (Output the character info 127);
  (Output the dimensions themselves 129);
  (Output the ligature/kern program 130);
  (Output the extensible character recipes 131);
  (Output the parameters 132)
```

This code is used in section 135.

121. A TFM file begins with 12 numbers that tell how big its subfiles are. We already know most of these numbers; for example, the number of distinct widths is *memory*[*width*] + 1, where the +1 accounts for the zero width that is always supposed to be present. But we still should compute the beginning and ending character codes (*bc* and ec), the number of header words *(Zh),* and the total number of words in the TFM file (*lf*).

```
(Globals in the outer block 5) +≡
bc: byte;  { the smallest character code in the font }
ec: byte;  {the largest character code in the font }
lh: byte;  { the number of words in the header block }
lf: 0..32767;  { the number of words in the entire TFM file }
not-found: boolean;  { has a font character been found? }
temp_width: fix-word;  { width being used to compute a check sum }
```

122. 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 122) ≡
  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>ecthen bc ← 1;
  incr (memory [width]); incr (memory [height]); incr (memory [depth]); incr (memory [italic]);
  lf ← 6 + lh + (ec − bc + 1) + memory [width] + memory [height] + memory [depth] + memory [italic] + nl +
      nk + ne + np;
```

This code is used in section 120.

123. **define out-size(#) ≡** *out*((#) **div** 256); *out*((#) **mod** 256)

(Output the twelve subfile sizes **123)** ≡

out-size (*lf*); **out-size (Zh); out-size (bc);** *out_size* **(ec);** *out_size*(**memory [width]);**
out-size (memory [height]); out-size (memory [depth]); out-size (memory [italic]); **out-size**(*nl*);
out-size (*nk*); ***out-size (ne); out-size (np);***

This code is used in section 120.

124. The routines that follow need a few temporary variables of different types.

(Globals in the outer block 5) +≡

j: 0.. *max_header_bytes*; { index into ***header-bytes*** }
p: pointer: { index into ***memory*** }
q: ***width . . italic;*** { **runs** through the list heads for dimensions }
par-ptr : 0 . . *max_param_words*; { runs through the parameters }

125. The header block follows the subfile sizes. The necessary information all appears in ***header-bytes,*** except that the design size and the seven-bit-safe flag must still be set.

(Output the header block **125** ⟩ ≡

if ¬*check_sum_specified* **then** (Compute the check sum **126);**
header-bytes [*design_size_loc*] ← ***design-size*** **div** *'100000000* ; { this works since ***design-size*** > ***0*** }
header_bytes[*design_size_loc* + 1] ← ***(design-size*** **div 200000) mod 256;**
header-bytes [***design-size-Zoc*** + 2] ← ***(design-size*** **div 256) mod 256;**
header-bytes [design-size-Zoc + 3] ← ***design-size*** **mod 256;**
if ¬*seven_unsafe* **then** *header_bytes*[*seven_flag_loc*] ← ***128;***
for *j* ← ***0*** **to** ***header-ptr*** − 1 **do** ***out (header-bytes*** [*j*]);

This code is used in section 120.

126. (Compute the check sum **126)** ≡

begin *c0* ← ***bc;*** ***cl*** ← ***ec;*** *c2* ← ***bc;*** *c3* ← ***ec;***
for c ← ***bc*** **to ec do**
if ***char,wd*** **[c] > 0 then**
begin ***temp-width*** ← ***memory*** [***char-wd*** [*c*]];
if ***design-units*** ≠ ***unity*** **then** ***temp-width*** ← *trunc*((*temp_width*/*design_units*) ∗ **1048576.0);**
temp-width ← ***temp-width*** + ***(c*** + ***4)*** ∗ 20000000 ; { this should be positive }
c0 ← (*c0* + *c0* + ***temp-width)*** **mod** ***255;*** ***cl*** ← ***(cl*** + ***cl*** + ***temp-width)*** **mod 253;**
c2 ← (*c2* + *c2* + ***temp-width)*** **mod 251;** *c3* ← (*c3* + *c3* + ***temp-width)*** **mod 247;**
end:
header-bytes [***check-sum-Zoc]*** t *c0*; *header_bytes*[*check_sum_loc* + 1] ← ***cl*** ;
header-bytes [check-sum-Zoc + 2] ← *c2* ; *header_bytes*[*check_sum_loc* + 3] ← *c3*;
end

This code is used in section 125.

127. The next block contains packed ***char-info.***

(Output the character info **127)** ≡

index[0] ← ***0;***
for c ← ***bc*** **to ec do**
begin ***out (index*** [***char-wd*** [*c*]]); ***out (index [char-ht*** [*c*]] ∗ **16** + ***index [char-dp*** [*c*]]);
out (index [*char_ic*[*c*]] ∗ ***4*** + ***char-tag*** [*c*]); ***out (char-remainder*** [*c*]);
end

This code is used in section 120.

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

```
procedure out_scaled(x : fix-word);  { outputs a scaled fix-word }
  var z: real;  { a number to output after conversion to fixed point }
    n: byte;  {the first byte after the sign}
    m: 0 . . 65535;  { the two least significant bytes }
  begin if abs(x/design_units)≥ 16.0 then
    begin print_ln('The relative dimension', x/ '4000000 : 1 : 3, ' is too large.');
    print ('  (Must be less than 16*designsize');
    if design-units ≠ unity then print( ' =', design-units/ '200000 : 1 : 3, ' designunits');
    print-Zn( ' )'); x + 0;
    end:
  if x < 0 then out (255)
  else out (0);
  if design-units = unity then
    begin if x < 0 then x ← x + '10000000~;
    n ← x div 200000; m ← x mod 200000;
    end
  else begin z ← (x/design_units) * 16.0;
    if z < 0 then z ← z + 256.0;
    n ← trunc(z); m ← trunc(65536.0 * (z − n));
    end:
  out(n): out(m div 256); out(m mod 256);
  end:
```

129. We have output the packed indices for individual characters. The scaled widths, heights, depths, and it alic correct ions are next.

```
⟨ Output the dimensions themselves 129 ⟩≡
  for q ← width to italic do
    begin out (0); out (0); out(O); out (0);  { output the zero word}
    p ← link [q];  { head of list }
    while p > 0 do
      begin out-scaled (memory [p]); p ← link [p];
      end:
    end:
```

This code is used in section 120.

130. ⟨Output the ligature/kern program 130⟩≡

```
  if nl > 0 then
    for Zig-ptr ← 0 to nl − 1 do
    . begin out(lig_kern[lig_ptr].b0); out(lig_kern[lig_ptr].b1); out(lig_kern[lig_ptr].b2);
      out(lig_kern[lig_ptr].b3);
      end:
  if nk > 0 then
    for krn_ptr ← 0 to nk − 1 do out-scaled (kern [krn_ptr])
```

This code is used in section 120.

131. (Output the extensible character recipes 131 ⟩ ≡

```
if ne > 0 then
  for c ← 0 to ne − 1 do
    begin out(exten[c].b0); out(exten[c].b1); out(exten[c].b2); out(exten[c].b3);
    end;
```

This code is used in section 120.

132. For our grand finale, we wind everything up by outputting the parameters.

(Output the parameters 132) ≡

```
for par-ptr ← 1 to np do
  begin if par-ptr = 1 then (Output the slant (param[1]) without scaling 133 ⟩
  else out-scaled (param [par-ptr]);
  end
```

This code is used in section 120.

133. (Output the slant (*param*[1]) without scaling 133 ⟩ ≡

```
begin if param[1] < 0 then
  begin param[1] ← param[1] + '10000000000; out((param[1] div '100000000) + 256 − 64);
  end
else out(param[1] div '100000000);
out((param[1] div 200000) mod 256); out((param[1] div 256) mod 256); out(param[1] mod 256);
end
```

This code is used in section 132.

134. 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.

```
procedure param_enter;
  begin ( Enter the parameter names 48 );
  end;
procedure name-enter; { enter all names and their equivalents }
  begin ( Enter all of the names and their equivalents, except the parameter names 47 ) ;
  param_enter;
  end;
procedure read_lig_kern;
  var krn_ptr: 0 .. 256; { an index into kern }
    c: byte; { runs through all character codes }
  begin ( Read ligature/kern list 94 );
  end;
procedure read-char-info;
  begin ( Read character info list 102);
  end;
procedure read-input;
  begin ( Read all the input 82 );
  end;
procedure corr_and_check;
  var c: byte; { runs through all character codes }
    Zig-ptr: 0 . . 511; { an index into Zig-kern }
    g: byte; {a character generated by the current character c}
  begin ( Correct and check the information 109 )
  end;
```

135. Here is where PLtoTF begins and ends.

```
begin initialize ;
name-enter;
read-input; print-Zn ('.');
corr-and-check:
(Do the output 120);
end.
```

136. System-dependent changes. This section should be replaced, if necessary, by changes to the program that are necessary to make PLtoTF 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.

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

The DVItype processor

(Version 2.8, August 1984)

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 DVItype utility program reads binary device-independent ("DVI") files that are produced by document compilers such as TEX, and converts them into symbolic form. This program has two chief purposes: (1) It can be used to determine whether a DVI file is valid or invalid, when diagnosing compiler errors; and (2) it serves as an example of a program that reads DVI files correctly, for system programmers who are developing DVI-related software.

Goal number (2) needs perhaps a bit more explanation. Programs for typesetting need to be especially careful about how they do arithmetic; if rounding errors accumulate, margins won't be straight, vertical rules won't line up, and so on. But if rounding is done everywhere, even in the midst of words, there will be uneven spacing between the letters. and that looks bad. Human eyes notice differences of a thousandth of an inch in the positioning of lines that are close together; on low resolution devices, where rounding produces effects four times as great as this, the problem is especially critical. Experience has shown that unusual care is needed even on high-resolution equipment: for example, a mistake in the sixth significant hexadecimal place of a constant once led to a difficult-to-find bug in some software for the Alphatype CRS, which has a resolution of 5333 pixels per inch (make that 5333.33333333 pixels per inch). The document compilers that generate DVI files make certain assumptions about the arithmetic that will be used by DVI-reading software, and if these assumptions are violated the results will be of inferior quality. Therefore the present program is intended as a guide to proper procedure in the critical places where a bit of subtlety is involved.

The first DVItype program was designed by David Fuchs in 1979, and it went through several versions on different computers as the format of DVI files was evolving to its present form.

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

```
define banner ≡ 'This␣is␣DVItype,␣Version␣2.8'  { printed when the program starts }
```

2. This program is written in standard Pascal, except where it is necessary to use extensions; for example, DVItype must read files whose names are dynamically specified, and that would be impossible in pure Pascal. All places where nonstandard constructions are used have been listed in the index under "system dependencies."

One of the extensions to standard Pascal that we shall deal with is the ability to move to a random place in a binary file; another is to determine the length of a binary file. Such extensions are not necessary for *reading DVI files, and they are not important for efficiency reasons either—an infrequently used program like DVItype does not have to be efficient. But they are included there because of DVItype's rôle as a model of a DVI reading routine, since other DVI processors ought to be highly efficient. If DVItype is being used with Pascals for which random file positioning is not efficiently available, the following definition should be changed from *true* to ***false;*** in such cases, DVItype will not include the optional feature that reads the postamble first.

Another extension is to use a default **case** as in TANGLE, WEAVE, etc.

```
define random-reading ≡ true  { should we skip around in the file? }
define othercases ≡ others:  { default for cases not listed explicitly }
define endcases ≡ end  { follows the default case in an extended case statement }
format othercases ≡ else
format endcases ≡ end
```

3. The binary input comes from *dvi_file*, and the symbolic output is written on Pascal's standard ***output*** file. The term ***print*** is used instead of ***write*** when this program writes on ***output,*** so that all such output could easily be redirected if desired.

```
define print(#) ≡ write(#)
define print_ln(#) ≡ write_ln(#)
program DVI_type(dvi_file, output);
  label ⟨ Labels in the outer block 4 ⟩
  const ( Constants in the outer block 5)
  type ( Types in the outer block 8 ⟩
  var ( Globals in the outer block 10 ⟩
  procedure initialize ; { this procedure gets things started properly}
    var i: integer; { loop index for initializations }
    begin print-Zn (banner);
    ( Set initial values 11)
    end;
```

4. If the program has to stop prematurely. it goes to the ***final-end***: Another label, ***done,*** is used when stopping normally.

```
define final-end = 9999 { label for the end of it all }
define done = 30 { go here when finished with a subtask }
( Labels in the outer block 4) ≡
  final-end, done:
```

This code is used in section 3.

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

```
( Constants in the outer block 5) ≡
  max_fonts = 100: { maximum number of distinct fonts per DVI file }
  max_widths = 10000; { maximum number of different characters among all fonts }
  line-length = 79; { bracketed lines of output will be at most this long}
  terminal-line-length. = 150;
      { maximum number of characters input in a single line of input from the terminal }
  stack-size = 100; { DVI files shouldn't push beyond this depth }
  name-size = 1000: { total length of all font file names }
  name_length = 50; { a file name shouldn't be longer than this }
```

This code is used in section 3.

6. Here are some macros for common programming idioms.

```
define incr(#) ≡ # ← # + 1 { increase a variable by unity}
define decr(#) ≡ # ← # − 1 { decrease a variable by unity }
define do-nothing ≡ { empty statement }
```

7. If the DVI file is badly malformed, the whole process must be aborted; DVItype will give up, after issuing an error message about the symptoms that were noticed.

Such errors might be discovered inside of subroutines inside of subroutines, so a procedure called ***jump-out*** has been introduced. This procedure. which simply transfers control to the label *final_end* at the end of the program. contains the only non-local **goto** statement in DVItype.

```
define abort(#) ≡
        begin print('␣'.#); jump-out;
        end
define bad_dvi(#) ≡ abort('Bad␣DVI␣file␣:␣'.#.'!')
procedure jump_out;
  begin goto final_end;
  end;
```

8. **The character set.** Like all programs written with the WEB system, DVItype can be used with any character set. But it uses ASCII code internally, because the programming for portable input-output is easier when a fixed internal code is used, and because DVI files use ASCIJ code for file names and certain other strings.

The next few sections of DVItype have therefore been copied from the analogous ones in the WEB system routines. They have been considerably simplified, since DVItype need not deal with the controversial ASCII codes less than '40. If such codes appear in the DVI file, they will be printed as question marks.

⟨ Types in the outer block 8 ⟩ ≡

ASCII-code = "␣" . . "~"; { a subrange of the integers }

See also sections 9 and 21.

This code is used in section 3.

9. The original Pascal compiler was designed in the late 60s, when six-bit character sets were common, so it did not make provision for lower case letters. Nowadays, of course, we need to deal with both upper and lower case alphabets in a convenient way, especially in a program like DVItype. So we shall assume that the Pascal system being used for DVItype has a character set containing at least the standard visible characters of ASCII code ("!" through "~").

Some Pascal compilers use the original name *char* for the data type associated with the characters in text files, while other Pascals consider char to be a 64-element subrange of a larger data type that has some other name. In order to accommodate this difference, we shall use the name *text_char* to stand for the data type of the characters in the output file. We shall also assume that *text_char* consists of the elements *chr*(*first_text_char*) through *chr*(*last_text_char*), inclusive. The following definitions should be adjusted if necessary.

define test-char ≡ ***char*** { the data type of characters in text files}

define *first-text-char* = 0 { ordinal number of the smallest element of *text-chur* }

define ***last-text-char*** = 127 { ordinal number of the largest element of *text-chur* }

⟨ Types in the outer block 8 ⟩ +≡

text-file = **packed file of** ***text-chur:***

10. The DVItype processor converts between ASCII code and the user's external character set by means of arrays *xord* and *xchr* that are analogous to Pascal's *ord* and ***chr*** functions.

⟨ Globals in the outer block 10 ⟩ ≡

xord: **array** [*text_char*] **of** *ASCII_code*: { specifies conversion of input characters }

xchr: **array [0 . . 255] of** ***text-char:*** { specifies conversion of output characters }

See also sections 22, 24. 25. 30, 33. 39. 41, 42, 45, 48, 57, 64, 67, 72, 73, 78, 96, 100, and 107.

This code is used in section 3.

11. Under our assumption that the visible characters of standard ASCII are all present, the following assignment statements initialize the ***xchr*** array properly, without needing any system-dependent changes.

```
( Set initial values 11) ≡
  for i ← 0 to 37 do xchr[i] ← ´?´;
  xchr[´40] ← ´␣´; xchr[´41] ← ´!´; xchr[´42] ← ´"´; xchr[´43] ← ´#´; xchr[´44] ← ´$´;
  xchr[´45] ← ´%´; xchr[´46] ← ´&´; xchr[´47] ← ´'´;
  xchr[´50] ← ´(´; xchr[´51] ← ´)´; xchr[´52] ← ´*´; xchr[´53] ← ´+´; xchr[´54] ← ´,´;
  xchr[´55] ← ´-´; xchr[´56] ← ´.´; xchr[´57] ← ´/´;
  xchr[´60] ← ´0´; xchr[´61] ← ´1´; xchr[´62] ← ´2´; xchr[´63] ← ´3´; xchr[´64] ← ´4´;
  xchr[´65] ← ´5´; xchr[´66] ← ´6´; xchr[´67] ← ´7´;
  xchr[´70] ← ´8´; xchr[´71] ← ´9´; xchr[´72] ← ´:´; xchr[´73] ← ´;´; xchr[´74] ← ´<´;
  xchr[´75] ← ´=´; xchr[´76] ← ´>´; xchr[´77] ← ´?´;
  xchr[´100] ← ´@´; xchr[´101] ← ´A´; xchr[´102] ← ´B´; xchr[´103] ← ´C´; xchr[´104] ← ´D´;
  xchr[´105] ← ´E´; xchr[´106] ← ´F´; xchr[´107] ← ´G´;
  xchr[´110] ← ´H´; xchr[´111] ← ´I´; xchr[´112] ← ´J´; xchr[´113] ← ´K´; xchr[´114] ← ´L´;
  xchr[´115] ← ´M´; xchr[´116] ← ´N´; xchr[´117] ← ´O´;
  xchr[´120] ← ´P´; xchr[´121] ← ´Q´; xchr[´122] ← ´R´; xchr[´123] ← ´S´; xchr[´124] ← ´T´;
  xchr[´125] ← ´U´; xchr[´126] ← ´V´; xchr[´127] ← ´W´;
  xchr[´130] ← ´X´; xchr[´131] ← ´Y´; xchr[´132] ← ´Z´; xchr[´133] ← ´[´; xchr[´134] ← ´\´;
  xchr[´135] ← ´]´; xchr[´136] ← ´^´; xchr[´137] ← ´_´;
  xchr[´140] ← ´`´; xchr[´141] ← ´a´; xchr[´142] ← ´b´; xchr[´143] ← ´c´; xchr[´144] ← ´d´;
  xchr[´145] ← ´e´; xchr[´146] ← ´f´; xchr[´147] ← ´g´;
  xchr[´150] ← ´h´; xchr[´151] ← ´i´; xchr[´152] ← ´j´; xchr[´153] ← ´k´; xchr[´154] ← ´l´;
  xchr[´155] ← ´m´; xchr[´156] ← ´n´; xchr[´157] ← ´o´;
  xchr[´160] ← ´p´; xchr[´161] ← ´q´; xchr[´162] ← ´r´; xchr[´163] ← ´s´; xchr[´164] ← ´t´;
  xchr[´165] ← ´u´; xchr[´166] ← ´v´; xchr[´167] ← ´w´;
  xchr[´170] ← ´x´; xchr[´171] ← ´y´; xchr[´172] ← ´z´; xchr[´173] ← ´{´; xchr[´174] ← ´|´;
  xchr[´175] ← ´}´; xchr[´176] ← ´~´;
  for i ← ´177 to 255 do xchr[i] ← ´?´;
```

See also sections 12, 31, 43, 58, 65. 68, 74. and 97.

This code is used in section 3.

12. The following system-independent, code makes the ***xord*** array contain a suitable inverse to the information in ***xchr.***

```
(Set initial values 11) +≡
  for i ← first_text_char to last_text_char do xord[chr(i)] ← ´40;
  for i ← "␣" to "~" do xord[xchr[i]] ← i;
```

13. Device-independent file format. Before we get into the details of DVItype, we need to know exactly what DVI files are. The form of such files was designed by David R. Fuchs in 1979. Almost any reasonable typesetting device can be driven by a program that takes DVI files as input, and dozens of such DVI-to-whatever programs have been written. Thus, it is possible to print the output of document compilers like TEX on many different kinds of equipment.

A DVI file is a stream of 8-bit bytes, which may be regarded as a series of commands in a machine-like language. The first byte of each command is the operation code, and this code is followed by zero or more bytes that provide parameters to the command. The parameters themselves may consist of several consecutive bytes; for example, the ***set-rule*** command has two parameters, each of which is four bytes long. Parameters are usually regarded as nonnegative integers; but four-byte-long parameters, and shorter parameters that denote distances, can be either positive or negative. Such parameters are given in two's complement notation. For example, a two-byte-long distance parameter has a value between -2^{15} and $2^{15}-1$.

A DVI file consists of a "preamble," followed by a sequence of one or more "pages," followed by a "postamble." The preamble is simply a ***pre*** command, with its parameters that define the dimensions used in the file; this must come first. Each "page" consists of a ***bop*** command, followed by any number of other commands that tell where characters are to be placed on a physical page, followed by an ***eop*** command. The pages appear in the order that they were generated, not in any particular numerical order. If we ignore ***nop*** commands and *fnt_def* commands (which are allowed between any two commands in the file), each ***eop*** command is immediately followed by a ***bop*** command, or by a ***post*** command; in the latter case, there are no more pages in the file, and the remaining bytes form the postamble. Further details about the postamble will be explained later.

Some parameters in DVI commands are "pointers." These are four-byte quantities that give the location number of some other byte in the file; the first byte is number 0, then comes number 1, and so on. For example, one of the parameters of a ***bop*** command points to the previous ***bop;*** this makes it feasible to read the pages in backwards order, in case the results are being directed to a device that stacks its output face up. Suppose the preamble of a DVI file occupies bytes 0 to 99. Now if the first page occupies bytes 100 to 999, say, and if the second page occupies bytes 1000 to 1999, then the ***bop*** that starts in byte 1000 points to 100 and the ***bop*** that starts in byte 2000 points to 1000. (The very first ***bop,*** i.e., the one that starts in byte 100, has a pointer of -1.)

14. The DVI format is intended to be both compact and easily interpreted by a machine. Compactness is achieved by making most of the information implicit instead of explicit. When a DVI-reading program reads the commands for a page, it keeps track of several quantities: (a) The current font ***f*** is an integer: this value is changed only by ***fnt*** and *fnt_num* commands. (b) The current position on the page is given by two numbers called the horizontal and vertical coordinates, ***h*** and v. Both coordinates are zero at the upper left corner of the page; moving to the right corresponds to increasing the horizontal coordinate, and moving down corresponds to increasing the vertical coordinate. Thus, the coordinates are essentially Cartesian, except that vertical directions are flipped; the Cartesian version of (h, v) would be $(h, -v)$. (c) The current spacing amounts are given by four numbers w, x, y, and z, where w and x are used for horizontal spacing and where y and z are used for vertical spacing. (d) There is a stack containing (h, v, w, x, y, z) values; the DVI commands ***push*** and ***pop*** are used to change the current level of operation. Note that the current font ***f*** is not pushed and popped; the stack contains only information about positioning.

The values of h, v, w, x, y, and z are signed integers having up to 32 bits, including the sign. Since they represent physical distances, there is a small unit of measurement such that increasing ***h*** by 1 means moving a certain tiny distance to the right. The actual unit of measurement is variable, as explained below.

15. Here is a list of all the commands that may appear in a DVI file. Each command is specified by its symbolic name (e.g., bop), its opcode byte (e.g., 139), and its parameters (if any). The parameters are followed by a bracketed number telling how many bytes they occupy; for example, '$p[4]$' means that parameter p is four bytes long.

set_char_0 0. Typeset character number 0 from font f such that the reference point of the character is at (h, v). Then increase **h** by the width of that character. Note that a character may have zero or negative width, so one cannot be sure that h will advance after this command; but **h** usually does increase.

set-char-1 through *set_char_127* (opcodes 1 to 127). Do the operations of *set_char_0*; but use the character whose number matches the opcode, instead of character 0.

set1 128 $c[1]$. Same as *set_char_0*, except that character number c is typeset. TEX82 uses this command for characters in the range $128 \leq c < 256$.

set2 129 $c[2]$. Same as *set1*, except that c is two bytes long, so it is in the range $0 \leq c < 65536$. TEX82 never uses this command, which is intended for processors that deal with oriental languages; but DVItype will allow character codes greater than 255, assuming that they all have the same width as the character whose code is c mod 256.

set3 **130** $c[3]$. Same as set1, except that c is three bytes long, so it can be as large as $2^{24} - 1$.

set4 131 $c[4]$. Same as *set1*, except that c is four bytes long, possibly even negative. Imagine that.

set-rule 132 $a[4]\, b[4]$. Typeset a solid black rectangle of height a and width b, with its bottom left corner at (h, v). Then set $h \leftarrow h + b$. If either $a \leq 0$ or $b \leq 0$, nothing should be typeset. Note that if $b < 0$, the value of h will decrease even though nothing else happens. Programs that typeset from DVI files should be careful to make the rules line up carefully with digitized characters, as explained in connection with the *rule_pixels* subroutine below.

put1 133 $c[1]$. Typeset character number c from font f such that the reference point of the character is at (h, v). (The 'put' commands are exactly like the 'set' commands, except that they simply put out a character or a rule without moving the reference point afterwards.)

put2 134 $c[2]$. Same as *set2*, except that h is not changed.

put3 135 $c[3]$. Same as *set3*, except that **h** is not changed.

put4 136 $c[4]$. Same as *set4*, except that h is not changed.

put-rule 137 $a[4]\, b[4]$. Same as ***set-rule,*** except that **h** is not changed.

nop 138. No operation, do nothing. Any number of *nop*s may occur between DVI commands, but a *nop* cannot be inserted between a command and its parameters or between two parameters.

bop 139 $c_0[4]\, c_1[4] \ldots c_9[4]\, p[4]$. Beginning of a page: Set $(h, v, w, x, y, z) \leftarrow (0, 0, 0, 0, 0, 0)$ and set the stack empty. Set the current font f to an undefined value. The ten c_i parameters can be used to identify pages, if a user wants to print only part of a DVI file; TEX82 gives them the values of \count0 ... \count9 at the time \shipout was invoked for this page. The parameter p points to the previous ***bop*** command in the file, where the first ***hop*** has $p = -1$.

eop 140. End of page: Print what you have read since the previous ***bop.*** At this point the stack should be empty. (The DVI-reading programs that drive most output devices will have kept a buffer of the material that appears on the page that has just ended. This material is largely, but not entirely, in order by v coordinate and (for fixed v) by **h** coordinate; so it usually needs to be sorted into some order that is appropriate for the device in question. DVItype does not do such sorting.)

push 141. Push the current values of (h, v, w, x, y, z) onto the top of the stack: do not change any of these values. Note that f is not pushed.

pop 142. Pop the top six values off of the stack and assign them to (h, v, w, x, y, z). The number of pops should never exceed the number of pushes, since it would be highly embarrassing if the stack were empty at the time of a *pop* command.

right1 143 $b[1]$. Set $h \leftarrow h +$ **b.** i.e., move right **b** units. The parameter is a signed number in two's complement notation, $-128 \leq$ **b** < 128: if **b** < 0, the reference point actually moves left.

right2 144 $b[2]$. Same as *right1*, except that b is a two-byte quantity in the range $-32768 \le b < 32768$.

right9 145 $b[3]$. Same as ***right1***, except that b is a three-byte quantity in the range $-2^{23} \le b < 2^{23}$.

right4 146 $b[4]$. Same as *right1*, except that b is a four-byte quantity in the range $-2^{31} \le b < 2^{31}$.

w0 147. Set $h \leftarrow h + w$; i.e., move right w units. With luck, this parameterless command will usually suffice, because the same kind of motion will occur several times in succession; the following commands explain how w gets particular values.

w1 148 $b[1]$. Set $w \leftarrow b$ and $h \leftarrow h + b$. The value of b is a signed quantity in two's complement notation. $-128 \le b < 128$. This command changes the current w spacing and moves right by b.

w2 149 $b[2]$. Same as *w1*, but b is a two-byte-long parameter, $-32768 \le b < 32768$.

w3 150 $b[3]$. Same as *w1*, but b is a three-byte-long parameter, $-2^{23} \le b < 2^{23}$.

w4 151 $b[4]$. Same as *w1*, but b is a four-byte-long parameter, $-2^{31} \le b < 2^{31}$.

x0 152. Set $h \leftarrow h + x$; i.e.. move right x units. The 'x' commands are like the 'w' commands except that they involve x instead of w.

x1 153 $b[1]$. Set $x \leftarrow b$ and $h \leftarrow h + b$. The value of b is a signed quantity in two's complement notation, $-128 \le b < 128$. This command changes the current x spacing and moves right by b.

x2 154 $b[2]$. Same as *x1*, but b is a two-byte-long parameter, $-32768 \le b < 32768$.

x3 155 $b[3]$. Same as ***xl.*** but b is a three-byte-long parameter, $-2^{23} \le b < 2^{23}$.

x4 156 $b[4]$. Same as xl . but b is a four-byte-long parameter, $-2^{31} \le b < 2^{31}$.

down1 157 $a[1]$. Set $v \leftarrow v + a$. i.e., move down a units. The parameter is a signed number in two's complement notation, $-128 \le a < 128$; if $a < 0$, the reference point actually moves up.

down& 158 $a[2]$. Same as *down1*. except that a is a two-byte quantity in the range $-32768 \le a < 32768$.

down3 159 $a[3]$. Same as *down1*, except that a is a three-byte quantity in the range $-2^{23} \le a < 2^{23}$.

down4 160 $a[4]$. Same as ***down1***, except that a is a four-byte quantity in the range $-2^{31} \le a < 2^{31}$.

y0 161. Set $v \leftarrow v + y$; i.e., move down y uuits. With luck, this parameterless command will usually suffice. because the same kind of motion will occur several times in succession: the following commands explain how y gets particular values.

yl 162 $a[1]$. Set $y \leftarrow a$ and $v \leftarrow v + a$. The value of a is a signed quantity in two's complement notation. $-128 \le a < 128$. This command changes the current y spacing and moves down by a.

y2 163 $a[2]$. Same as *y1*, but a is a two-byte-long parameter, $-32768 \le a < 32768$.

y3 164 $a[3]$. Same as *y1*. but a is a three-byte-long parameter. $-2^{23} \le a < 2^{23}$.

y4 165 $a[4]$. Same as *y1*. but a is a four-byte-long parameter, $-2^{31} \le a < 2^{31}$.

z0 166. Set $v \leftarrow v + z$; i.e., move down z units. The 'z' commands are like the 'y' commands except that they involve z instead of y.

z1 167 $a[1]$. set $z \leftarrow a$ and $v \leftarrow v + a$. The value of a is a signed quantity in two's complement notation. $-128 \le a < 128$. This command changes the current z spacing and moves down by a.

z2 168 $a[2]$. Same as *z1*, but a is a two-byte-long parameter, $-32768 \le a < 32768$.

23 169 $a[3]$. Same as *z1*, but a is a three-byte-long parameter. $-2^{23} \le a < 2^{23}$.

z4 170 $a[4]$. Same as *z1*, but a is a four-byte-long parameter. $-2^{31} \le a < 2^{31}$.

fnt_num_0 171. Set $f \leftarrow 0$. Font 0 must, previously have been defined by a *fnt_def* instruction. as explained below.

fnt_num_1 through *fnt_num_63* (opcodes 172 to 234). Set $f \leftarrow 1, \ldots, f \leftarrow 63$. respectively.

fnt1 2 3 5 $k[1]$. Set $f \leftarrow k$. TEX82 uses this command for font numbers in the range $64 \le k < 256$.

fnt2 236 $k[2]$. Same as *fnt1*. except that k is two bytes long. so it is in the range $0 \le k < 65536$. TEX82 never generates this command, but large font numbers may prove useful for specifications of color or texture, or they may be used for special fonts that have fixed numbers in some external coding scheme.

fnt3 237 $k[3]$. Same as *fnt1*, except that k is three bytes long, so it can be as large as $2^{24} - 1$.

fnt4 238 $k[4]$. Same as *fnt1*, except that k is four bytes long; this is for the really big font numbers (and for the negative ones).

xxx1 **239** $k[1]$ $x[k]$. This command is undefined in general; it functions as a $(k + 2)$-byte *nop* unless special DVI-reading programs are being used. TEX82 generates *xxx1* when a short enough \special appears, setting k to the number of bytes being sent. It is recommended that x be a string having the form of a keyword followed by possible parameters relevant to that keyword.

xxx2 240 $k[2]$ $x[k]$. Like *xxx1*, but $0 \le k < 65536$.

xxx3 241 $k[3]$ $x[k]$. Like *xxx1*, but $0 \le k < 2^{24}$.

xxx4 242 $k[4]$ $x[k]$. Like *xxx1*, but k can be ridiculously large. TEX82 uses *xxx4* when xxx1 would be incorrect.

fnt_def1 243 k[l] $c[4]$ $s[4]$ $d[4]$ $a[1]$ $l[1]$ $n[a + l]$. Define font k, where $0 \le k < 256$; font definitions will be explained shortly.

fnt_def2 244 $k[2]$ $c[4]$ $s[4]$ $d[4]$ $a[1]$ $l[1]$ $n[a + l]$. Define font k, where $0 \le k < 65536$.

fnt_def3 245 $k[3]$ $c[4]$ $s[4]$ $d[4]$ $a[1]$ $l[1]$ $n[a + l]$. Define font k, where $0 \le k < 2^{24}$.

fnt_def4 246 $k[4]$ $c[4]$ $s[4]$ $d[4]$ $a[1]$ $l[1]$ $n[a + l]$. Define font k, where $-2^{31} \le k < 2^{31}$.

pre **247** $i[1]$ *num* $[4]$ *den* $[4]$ *mag* $[4]$ $k[1]$ $x[k]$. Beginning of the preamble; this must come at the very beginning of the file. Parameters i, *num, den, mug, k,* and x are explained below.

post 248. Beginning of the postamble, see below.

post-post 249. Ending of the postamble, see below.

Commands 250-255 are undefined at the present time.

16. **define** ***set-char-0*** = 0 { typeset character 0 and move right }
define *set1* = 128 { typeset a character and move right }
define ***set-rule*** = 132 { typeset a rule and move right }
define *put1* = 133 { typeset a character }
define ***put-rule*** = 137 { typeset a rule }
define ***nop*** = 138 { no operation }
define ***bop*** = 139 { beginning of page }
define ***eop*** = 140 { ending of page }
define ***push*** = 141 { save the current positions }
define ***pop*** = 142 { restore previous positions }
define *right1* = 143 { move right }
define *w0* = 147 { move right by w }
define *w1* = 148 { move right and set w }
define **x0** = 152 { move right by x }
define ***xl*** = 153 { move right and set x }
define ***down1*** = 157 { move down }
define *y0* = 161 { move down by y }
define *y1* = **162** { move down and set y }
define *z0* = **166** { move down by z }
define *z1* = 167 { move down and set z }
define ***fnt-num-0*** = 171 { set current font to 0 }
define ***fntl*** = 235 { set current font }
define *xxx1* = 239 { extension to DVI primitives }
define *xxx4* = 242 { potentially long extension to DVI primitives }
define ***fnt-defl*** = 243 { define the meaning of a font number }
define ***pre*** = 247 { preamble }
define ***post*** = 248 { postamble beginning }
define ***post-post*** = 249 { postamble ending }
define ***undefined-commands*** $\equiv 250, 251, 252, 253, 254, 255$

17. The preamble contains basic information about the file as a whole. As stated above. there are six parameters:

$$i[1]\ num[4]\ den[4]\ mag[4]\ k[1]\ x[k].$$

The i byte identifies DVI format: currently this byte is always set to 2. (Some day we will set i = 3, when DVI format makes another incompatible change-perhaps in 1992.)

The next two parameters, ***num*** and ***den,*** are positive integers that define the units of measurement; they are the numerator and denominator of a fraction by which all dimensions in the DVI file could be multiplied in order to get lengths in units of 10^{-7} meters. (For example, there are exactly 7227 TEX points in 254 centimeters, and TEX82 works with scaled points where there are 2^{16} sp in a point, so TEX82 sets ***num*** = 25400000 and ***den*** = $7227 \cdot 2^{16} = 473628672$.)

The ***mug*** parameter is what TEX82 calls **\mag**, i.e., 1000 times the desired magnification. The actual fraction by which dimensions are multiplied is therefore $mn/1000d$. Note that if a TEX source document does not call for any 'true' dimensions, and if you change it only by specifying a different \mag setting, the DVI file that TEX creates will be completely unchanged except for the value of ***mug*** in the preamble and postamble. (Fancy DVI-reading programs allow users to override the ***mug*** setting when a DVI file is being printed.)

Finally, k and x allow the DVI writer to include a comment, which is not interpreted further. The length of comment x is k, where $0 \le k < 256$.

define ***id-byte*** = 2 { identifies the kind of DVI files described here }

18. Font definitions for a given font number k contain further parameters

$$c[4]\ s[4]\ d[4]\ a[1]\ l[1]\ n[a+l].$$

The four-byte value c is the check sum that TEX (or whatever program generated the DVI file) found in the TFM file for this font; c should match the check sum of the font found by programs that read this DVI file.

Parameter s contains a fixed-point scale factor that is applied to the character widths in font **k;** font dimensions in TFM files and other font files are relative to this quantity, which is always positive and less than 2^{27}. It is given in the same units as the other dimensions of the DVI file. Parameter **d** is similar to s: it is the "design size," and it is given in DVI units that have not been corrected for the magnification **mug** found in the preamble. Thus, font **k** is to be used at **mug** . $s/1000d$ times its normal size.

The remaining part of a font definition gives the external name of the font, which is an ASCII string of length **a** + l. The number a is the length of the "area" or directory, and l is the length of the font name itself; the standard local system font area is supposed to be used when **a** = 0. The n field contains the area in its first **a** bytes.

Font definitions must appear before the first, use of a particular font number. Once font **k** is defined, it must not be defined again: however, we shall see below that font definitions appear in the postamble as well as in the pages. so in this sense each font number is defined exactly twice, if at all. Like **nop** commands and xxx commands, font definitions can appear before the first **bop,** or between an **eop** and a **bop.**

19. The last page in a DVI file is followed by **post;** this command introduces the postamble, which summarizes important facts that TEX has accumulated about the file, making it possible to print subsets of the data with reasonable efficiency. The postamble has the form

post $p[4]\ num[4]\ den[4]\ mag[4]\ l[4]\ u[4]\ s[2]\ t[2]$
⟨ font definitions ⟩
post-post $q[4]\ i[1]\ 223$'s$[\geq 4]$

Here **p** is a pointer to the final **bop** in the file. The next three parameters, **num, den,** and **mug,** are duplicates of the quantities that appeared in the preamble.

Parameters l and u give respectively the height-plus-depth of the tallest page and the width of the widest page. in the same units as other dimensions of the file. These numbers might be used by a DVI-reading program to position individual "pages" on large sheets of filrn or paper: however. the standard convention for output on normal size paper is to position each page so that the upper left-hand corner is exactly one inch from the left and the top. Experience has shown that it is unwise to design DVI-to-printer software that attempts cleverly to center the output: a fixed position of the upper left corner is easiest for users to understand and to work with. Therefore l and u arc often ignored.

Parameter s is the maximum stack depth (i.e., the largest excess of **push** commands over **pop** commands) needed to process this file. Then comes t. the total number of pages **(bop** commands) present.

The postamblc continues with font definitions. which are any number of *fnt_def* commands as described above, -possibly interspersed with *nop* commands. Each font number that is used in the DVI file must be defined exactly twice: Once before it is first selected by a *fnt* command, and once in the postamble.

20. The last part of the postamble, following the ***post-post*** byte that signifies the end of the font definitions, contains ***q,*** a pointer to the ***post*** command that started the postamble. An identification byte, i, comes next; this currently equals 2, as in the preamble.

The i byte is followed by four or more bytes that are all equal to the decimal number 223 (i.e., '337 in octal). TEX puts out four to seven of these trailing bytes, until the total length of the file is a multiple of four bytes, since this works out best on machines that pack four bytes per word; but any number of 223's is allowed, as long as there are at least four of them. In effect, 223 is a sort of signature that is added at the very end.

This curious way to finish off a DVI file makes it feasible for DVI-reading programs to find the postamble first, on most computers, even though TEX wants to write the postamble last. Most operating systems permit random access to individual words or bytes of a file, so the DVI reader can start at the end and skip backwards over the 223's until finding the identification byte. Then it can back up four bytes, read q, and move to byte q of the file. This byte should, of course, contain the value 248 ***(post);*** now the postamble can be read, so the DVI reader discovers all the information needed for typesetting the pages. Note that it is also possible to skip through the DVI file at reasonably high speed to locate a particular page, if that proves desirable. This saves a lot of time, since DVI files used in production jobs tend to be large.

Unfortunately, however, standard Pascal does not include the ability to access a random position in a file, or even to determine the length of a file. Almost all systems nowadays provide the necessary capabilities, so DVI format has been designed to work most efficiently with modern operating systems. As noted above, `DVItype` will limit itself to the restrictions of standard Pascal if ***random-reading*** is defined to be ***false.***

21. Input from binary files. We have seen that a DVI file is a sequence of 8-bit bytes. The bytes appear physically in what is called a **packed** file **of** 0 . . 255 in Pascal lingo.

Packing is system dependent, and many Pascal systems fail to implement such files in a sensible way (at least, from the viewpoint of producing good production software). For example, some systems treat all byte-oriented files as text, looking far end-of-line marks and such things. Therefore some system-dependent code is often needed to deal with binary files? even though most of the program in this section of DVItype is written in standard Pascal.

One common way to solve the problem *is* to consider files of ***integer*** numbers, and to convert an integer in the range $-2^{31} \le x < 2^{31}$ to a sequence of four bytes (a, *b,* c, *d*) using the following code, which avoids the controversial integer division of negative numbers:

```
if x ≥ 0 then a ← x div '100000000
else begin x ← (x + '10000000000) + '10000000000; a ← x div '100000000 + 128;
  end
x ← x mod '100000000 ;
b ← x div 200000; x ← x mod 200000;
c ← x div '400; d ← x mod '400;
```

The four bytes are then kept in a buffer and output one by one. (On 36-bit computers, an additional division by 16 is necessary at the beginning. Another way to separate an integer into four bytes is to use/abuse Pascal's variant records, storing an integer and retrieving bytes that are packed in the same place; caveat ***implementor!)*** It is also desirable in some cases to read a hundred or so integers at a time, maintaining a larger buffer.

We shall stick to simple Pascal in this program, for reasons of clarity, even if such simplicity is sometimes unrealistic.

```
⟨ Types in the outer block 8 ⟩ +≡
  eight-bits = 0 . . 255; { unsigned one-byte quantity }
  byte-file = packed file of eight-bits; { files that contain binary data}
```

22. The program deals with two binary file variables: ***dvi-file*** is the main input file that we are translating into symbolic form, and ***tfm-file*** is the current font metric file from which character-width information is being read.

```
⟨ Globals in the outer block 10 ⟩ +≡
dvi-file : byte-file ; { the stuff we are DVItyping }
tfm-file : byte-file ; { a font metric file }
```

23. To prepare these files for input, we ***reset*** them. An extension of Pascal is needed in the ***case*** of ***tfm-file ,*** since we want to associate it with external files whose names are specified dynamically (i.e., not known at compile time). The following code assumes that ● ***reset(f, s)*** does this, when *f* is a file variable and *s* is a string variable that specifies the file name. If *eof (f)* is true immediately after ***reset*** *(f, s)* has acted, we assume that no file named *s* is accessible.

```
procedure open_dvi_file; { prepares to read packed bytes in dvi_file }
  begin reset (dvi-file); cur_loc ← 0;
  end;
procedure open_tfm_file; { prepares to read packed bytes in tfm-file }
  begin reset ( tfm_file, cur-name);
  end;
```

24. If you looked carefully at the preceding code, you probably asked, "What are *cur_loc* and ***cur-name?***" Good question. They're global variables: *cur_loc* is the number of the byte about to be read next from *dvi_file*, and ***cur-name*** is a string variable that will be set to the current font metric file name before *open_tfm_file* is called.

⟨ Globals in the outer block 10 ⟩ +≡
cur-lot: integer; { where we are about to look, in *dvi_file* }
cur-name : **packed array** [1 .. ***name-length]*** **of** ***char;*** { external name, with no lower case letters }

25. It turns out to be convenient to read four bytes at a time, when we are inputting from TFM files. The input goes into global variables *b0*, *b1*, b2, and *b3*, with *b0* getting the first byte and *b3* the fourth.

⟨ Globals in the outer block 10 ⟩ +≡
b0, *b1*, ***b2, b3: eight-bits;*** { four bytes input at once }

26. The ***read-tfm-word*** procedure sets *b0* through ***b3*** to the next four bytes in the current TFM file.

```
procedure read-tfm-word;
  begin read( tfm-file, b0 ); read (tfm-file, bl); read( tfm-file, b2); read (tfm-file, b3);
  end;
```

27. We shall use another set of simple functions to read the next byte or bytes from *dvi_file*. There are seven possibilities, each of which is treated as a separate function in order to minimize the overhead for subroutine calls.

```
function get-byte: integer;  { returns the next byte, unsigned }
  var b: eight-bits:
  begin if eof(dvi_file) then get-byte ← 0
  else begin read (dvi-file, b); incr (cur,Zoc); get-byte ← b;
    end;
  end:
function signed-byte: integer;  { returns the next byte, signed }
  var b: eight-bits;
  begin read ( dvi-file, b): incr(cur-Zoc);
  if b < 128 then signed-byte ← b else signed-byte ← b − 256;
  end:
function get-two-bytes: integer;  { returns the next two bytes, unsigned }
  var a, b: eight-bits:
  begin read(dvi_file, a): read(dvi_file, b); cur-Zoc ← cur-Zoc + 2; get-two-bytes ← a * 256 + b;
  end:
function signed-paw: integer;  { returns the next two bytes, signed }
  var a. b: eight-bits:
  begin read (dvi-file . a); read (dvi-file , b); cur-Zoc ← cur-Zoc + 2;
  if a < 128 then signed-pair ← a * 256 + b
  else signed-pair ← (a − 256) * 256 + b;
  end:
function get-three-bytes: integer:  { returns the next three bytes, unsigned }
  var a. b, c: eight-bits;
  begin read (dvi-file . a); read (dvi_file , b); read (dvi_file , c): cur-Zoc ← cur-Zoc + 3;
  get-three-bytes ← (a * 256 + b) * 256 + c:
  end:
function signed-trio: integer:  { returns the next three bytes, signed}
  var a, b, c: eight-bits;
  begin read (dvi-file , a): read ( dvi_file , b): read (dvi-file. c): cur-Zoc ← cur_loc + 3:
  if a < 128 then signed_trio ← (a * 256 + b) * 256 + c
  else signed-trio ← ((a − 256) * 256 + b) * 256 + c;
  end:
function signed-quad: integer;  { returns the next four bytes, signed }
  var a. b. c. d: eight_bits;
  begin read ( dvi_file, a); read ( dvi-file, b); read ( dvi_file, c); read ( dvi_file, d); cur-Zoc ← cur-Zoc + 4;
  if a < 128 then signed-quad ← ((a * 256 + b) * 256 + c) * 256 + d
  else signed quad ← (((a − 256) * 256 + b) * 256 + c) * 256 + d:
  end:
```

28. Finally we come to the routines that are used only if ***random-reading*** is ***true.*** The driver program below needs two such routines: ***dvi-length*** should compute the total number of bytes in ***dvi-file,*** possibly also causing *eof* (*dvi_file*) to be true; and ***move-to-byte(n)*** should position ***dvi-file*** so that the next ***get-byte*** will read byte ***n,*** starting with n = 0 for the first byte in the file.

Such routines are, of course, highly system dependent. They are implemented here in terms of two assumed system routines called ***set-pos*** and ***cur-pos.*** The call *set_pos*(*f*, ***n)*** moves to item n in file *f*, unless ***n*** is negative or larger than the total number of items in *f*; in the latter case, *set_pos*(*f*, ***n)*** moves to the end of file *f*. The call *cur_pos*(*f*) gives the total number of items in *f*, if *eof*(*f*) is true; ***we*** use ***cur-pos*** only in such a situation.

```
function dvi-length: integer;
  begin set-pos (dvi-file, – 1); dvi-length ← cur-pos (dvi-file);
  end;
procedure move-to-byte(n : integer);
  begin set-pos (dvi-file , n); cur-Zoc ← n;
  end;
```

29. Reading the font information. DVI file format does not include information about character widths, since that would tend to make the files a lot longer. But a program that reads a DVI file is supposed to know the widths of the characters that appear in ***set-char*** commands. Therefore DVItype looks at the font metric (TFM) files for the fonts that are involved.

The character-width data appears also in other files (e.g., in GF files that specify bit patterns for digitized characters); thus, it is usually possible for DVI reading programs to get by with accessing only one file per font. DVItype has a comparatively easy task in this regard, since it needs only a few words of information from each font; other DVI-to-printer programs may have to go to some pains to deal with complications that arise when a large number of large font files all need to be accessed simultaneously.

30. For purposes of this program, we need to know only two things about a given character c in a given font f: (1) Is c a legal character in f? (2) If so, what is the width of c? We also need to know the symbolic name of each font, so it can be printed out, and we need to know the approximate size of inter-word spaces in each font.

The answers to these questions appear implicitly in the following data structures. The current number of known fonts is nf. Each known font has an internal number f, where $0 \le f < nf$; the external number of this font, i.e., its font identification number in the DVI file, is $font\_num[f]$, and the external name of this font is the string that occupies positions ***font-name*** $[f]$ through ***font-name*** $[f+1]-1$ of the array ***names.*** The latter array consists of ***ASCII-code*** characters, and ***font-name*** $[nf]$ is its first unoccupied position. A horizontal motion in the range ***-4 *font-space [f] < h < font-space*** $[f]$ will be treated as a 'kern' that is not indicated in the printouts that DVItype produces between brackets. The legal characters run from $font\_bc[f]$ to $font\_ec[f]$, inclusive; more precisely, a. given character c is valid in font f if and only if $font\_bc[f] \le c \le font\_ec[f]$ and ***char-width(f)(c)*** $\neq$ ***invalid-width.*** Finally, ***char-width (f)(c) = width [width-base*** $[f]$ + c], and ***width,ptr*** is the first unused position of the ***width*** array.

define *char-width-end* (#) ≡ #]
define *char-width* (#) ≡ ***width*** [*width_base*[#] + ***char-width-end***
define *invalid_width* ≡ **17777777777**

⟨ Globals in the outer block 10 ⟩ +≡
font-num: **array** [0 . . ***max-fonts]*** **of** ***integer;*** { external font numbers }
font_name: **array [0 . .** ***max-fonts]*** **of** ***0 . . name-size;*** { starting positions of external font names }
names: **array [0 . .** ***name-size]*** **of** ***ASCII-code;*** { characters of names }
font-checksum: **array** ***[0 . . max-fonts]*** **of** ***integer;*** { check sums }
font-scaled-size: **array** ***[0 . . max-fonts]*** **of** ***integer;*** { scale factors }
font-design-size: **array [0 . .** ***max-fonts]*** **of** ***integer;*** { design sizes }
font-space: **array [0 . .** ***max-fonts]*** **of** ***integer;*** { boundary between "small" and "large" spaces }
font-bc: **array [0 . .** ***max-fonts]*** **of** ***integer;*** { beginning characters in fonts }
font-ec: **array [0 . .** ***max-fonts]*** **of** ***integer;*** { ending characters in fonts }
width-base: **array [0 . .** ***max-fonts]*** **of** ***integer;*** { index into ***width*** table }
width: **array [0 . .** *max_widths*] **of** ***integer;*** { character widths, in DVI units }
nf: ***0.. max-fonts;*** { the number of known fonts }
width_ptr: ***0 . . max-widths;*** { the number of known character widths }

31. ⟨ Set initial values 11 ⟩ +≡
nf ← ***0; width,ptr*** ← ***0; font-name[O]*** ← ***0;*** *font_space*[0] ← ***0;***

32. It is, of course, a simple matter to print the name of a given font.

```
procedure print-font (f : integer); { f is an internal font number }
  var k: 0 .. name-size; { index into names }
  begin if f = nf then print( UNDEFINED! ´)
  else begin for k ← font-name[f] to font-name[f + 1] − 1 do print(xchr[names[k]]);
    end;
  end;
```

33. An auxiliary array ***in-width*** is used to hold the widths as they are input. The global variable ***tfm-check-sum*** is set to the check sum that appears in the current TFM file.

```
( Globals in the outer block 10 ⟩ +≡
in-width: array [0 .. 255] of integer; { TFM width data in DVI units}
tfm-check-sum: integer; { check sum found in tfm_file }
```

34. Here is a procedure that absorbs the necessary information from a TFM file, assuming that the file has just been successfully reset so that we are ready to read its first byte. (A complete description of TFM file format appears in the documentation of TFtoPL and will not be repeated here.) The procedure does not check the TFM file for validity, nor does it give explicit information about what is wrong with a TFM file that proves to be invalid; DVI-reading programs need not do this, since TFM files are almost always valid, and since the TFtoPL utility program has been specifically designed to diagnose TFM errors. The procedure simply returns ***false*** if it detects anything amiss in the TFM data.

There is a parameter, z, which represents the scaling factor being used to compute the font dimensions; it must be in the range $0 < z < 2^{27}$.

```
function in_TFM (z: integer): boolean; { input TFM data or return false }
  label 9997. { go here when the format is bad }
    9998, { go here when the information cannot be loaded }
    9999; { go here to exit }
  var k: integer; { index for loops }
    Zh: integer; { length of the header data, in four-byte words}
    nw : integer; { number of words in the width table }
    wp: 0.. max-widths; { new value of width-ptr after successful input }
    alpha, beta: integer; { quantities used in the scaling computation }
  begin ( Read past the header data; goto 9997 if there is a problem 35 ⟩;
  ( Store character-width indices at the end of the width table 36);
  (Read and convert the width values, setting up the in-width table 37);
  ( Move the widths from in-width to width, and append pixel-width values 40);
  width-ptr ← wp: in- TFM ← true; goto 9999;
9997: print-Zn (´---not␣loaded,␣TFM␣file␣is␣bad´);
9998: in_TFM ← false;
9999: end;
```

35. (Read past the header data; **goto** 9997 if there is a problem 35) ≡

```
read-tfm-word; lh ← b2 * 256 + b3; read-tfm-word; font_bc[nf] + b0 * 256 + bl ;
font_ec[nf] ← b2 * 256 + b3;
if font-ec[nf] < font-bc[nf] then font_bc[nf] ← font-ec[nf] + 1;
if width-ptr + font-ec[nf] − font_bc[nf] + 1 > max_widths then
  begin print_ln ('---not␣loaded,␣DVItype␣needs␣larger␣width␣table'); goto 9998;
  end;
wp t width-ptr + font_ec[nf] − font_bc[nf] + 1; read,tfm-word; nw ← b0 * 256 + bl ;
if (nw = 0) V (nw > 256) then goto 9997;
for k ← 1 to 3 + lh do
  begin if eof (tfm_file) then goto 9997;
  read-tfm-word;
  if k = 4 then
    if b0 < 128 then tfm-check-sum ← ((b0 * 256 + bl ) * 256 + b2) * 256 + b3
    else tfm-check-sum ← (((b0 − 256) * 256 + b1) * 256 + b2) * 256 + b3;
  e n d ;
```

This code is used in section 34.

36. (Store character-width indices at the end of the ***width*** table 36) ≡

```
if wp > 0 then
  for k ← width-ptr to wp − 1 do
    begin read-tfm-word;
    if b0 > nw then goto 9997:
    width[k] + b0;
    end:
```

This code is used in section 34.

37. The most important part of *in_TFM* is the width computation, which involves multiplying the relative widths in the TFM file by the scaling factor in the DVI file. This fixed-point multiplication must be done with precisely the same accuracy by all DVI-reading programs, in order to validate the assumptions made by DVI-writing programs like TEX82.

Let us therefore summarize what needs to be done. Each width in a TFM file appears as a four-byte quantity called a ***fix-word.*** A ***fix-word*** whose respective bytes are (a, ***b,*** c, ***d***) represents the number

$$x = \begin{cases} b \cdot 2^{-4} + c \cdot 2^{-12} + d \cdot 2^{-20}, & \text{if a = 0;} \\ -16 + b \cdot 2^{-4} + c \cdot 2^{-12} + d \cdot 2^{-20}, & \text{if } \boldsymbol{a = 255.} \end{cases}$$

(No other choices of a are allowed, since the magnitude of a TFM dimension must be less than 16.) We want to multiply this quantity by the integer z, which is known to be less than 2^{27}. Let $\alpha = 16z$. If $z < 2^{23}$, the individual multiplications $\boldsymbol{b} \cdot z, c \cdot z, d \cdot z$ cannot overflow; otherwise we will divide z by 2, 4, 8, or 16, to obtain a multiplier less than 2^{23}, and we can compensate for this later. If z has thereby been replaced by $z' = z/2^e$, let $\beta = 2^{4-e}$; we shall compute

$$\lfloor (b + \boldsymbol{c} \cdot 2^{-8} + \boldsymbol{d} \cdot 2^{-16}) z'/\beta \rfloor$$

if $a = 0$, or the same quantity minus α if a = 255. This calculation must be done exactly, for the reasons stated above: the following program does the job in a system-independent way, assuming that arithmetic is exact on numbers less than 2^{31} in magnitude.

```
⟨ Read and convert the width values, setting up the in-width table 37 ⟩≡
  ⟨ Replace z by z' and compute a, β 38 ⟩;
  for k ← 0 to nw − 1 do
    begin read-tfm-word: in_width[k] ← (((((b3 * z) div '400) + (b2 * z)) div '400) + (b1 * z)) div beta;
    if b0 > 0 then
      if b0 < 255 then goto 9997
      else in-width[k] ← in_width[k] − alpha;
    end
```

This code is used in section 34.

```
38. ⟨ Replace z by z' and compute α, β 38 ⟩≡
  begin alpha ← 16 * z; beta ← 16;
  while z ≥ '40000000 do
    begin z ← z div 2; beta ← beta div 2;
    end;
  end
```

This code is used in section 37.

39. A DVI-reading program usually works with font files instead of TFM files, so **DVItype** is atypical in that respect. Font files should, however, contain exactly the same character width data that is found in the corresponding **TFM**s; check sums are used to help ensure this. In addition, font. files usually also contain the widths of characters in pixels, since the device-independent character widths of TFM files are generally not perfect multiples of pixels.

The ***pixelwidth*** array contains this information; when ***width[k]*** is the device-independent width of some character in DVI units, ***pixel-width*** $[k]$ is the corresponding width of that character in an actual font. The macro ***char-pixel-width*** is set up to be analogous to ***char-width.***

```
define char-pixel-width (#) ≡ pixel-width [ width-base [#] + char-width-end
```

```
⟨ Globals in the outer block 10 ⟩ +≡
pixel-width: array [0 .. max_widths] of integer;   { actual character widths, in pixels }
conv : real;   { converts DVI units to pixels }
true-conv: real;   { converts unmagnified DVI units to pixels }
numerator, denominator: integer;   { stated conversion ratio }
mag : integer;   { magnification factor times 1000 }
```

40. The following code computes pixel widths by simply rounding the TFM widths to the nearest integer number of pixels. based on the conversion factor *conv* that converts DVI units to pixels. However, such a simple formula will not be valid for all fonts, and it will often give results that are off by ±1 when a low-resolution font has been carefully hand-fitted. For example, a font designer often wants to make the letter 'm' a pixel wider or narrower in order to make the font appear more consistent. DVI-to-printer programs should therefore input the correct pixel width information from font files whenever there is a chance that it may differ. A warning message may also be desirable in the case that at least one character is found whose pixel width differs from *conv* * ***width*** by more than a full pixel.

```
define pixel_round (#) ≡ round (conv * (#))
```

```
( Move the widths from in-width to width, and append pixelwidth values 40) ≡
  if in_width[0] ≠ 0 then goto 9997;   { the first width should be zero }
  width-base[nf] ← width-ptr − font_bc[nf];
  if wp > 0 then
    for k ← width-ptr to wp − 1 do
      if width [k] = 0 then
        begin width [k] ← invalid-width; pixel-width [k] ← 0;
        end
      else begin width [k] ← in-width [ width [k]]; pixel-width [k] ← pixel_round (width [k]);
        end
```

This code is used in section 34.

41. **Optional modes of output.** DVItype will print different quantities of information based on some options that the user must specify: The ***out-mode*** level is set to one of four values ***(errors-only, terse, verbose, the-works),*** giving different degrees of output; and the typeout can be confined to a restricted subset of the pages by specifying the desired starting page and the maximum number of pages. Furthermore there is an option to specify the resolution of an assumed discrete output device, so that pixel-oriented calculations will be shown; and there is an option to override the magnification factor that is stated in the DVI file.

The starting page is specified by giving a sequence of 1 to 10 numbers or asterisks separated by dots. For example, the specification '1.*.-5' can be used to refer to a page output by TEX when \count0 = 1 and \count2 = -5. (Recall that ***bop*** commands in a DVI file are followed by ten 'count' values.) An asterisk matches any number, so the '*' in '1.*.-5' means that \count 1 is ignored when specifying the first page. If several pages match the given specification, DVItype will begin with the earliest such page in the file. The default specification '*' (which matches all pages) therefore denotes the page at the beginning of the file.

When DVItype begins, it engages the user in a brief dialog so that the options will be specified. This part of DVItype requires nonstandard Pascal constructions to handle the online interaction; so it may be preferable in some cases to omit the dialog and simply to stick to the default options ***(out-mode = the-works,*** starting page '*', ***max-pages*** = 1000000, ***resolution*** = 240.0, ***new-mug*** = 0). On other hand, the system-dependent routines that are needed are not complicated, so it will not be terribly difficult to introduce them.

```
define errors-only = 0 { value of out-mode when minimal printing occurs }
define terse = 1 { value of out-mode for abbreviated output }
define verbose = 2 { value of out-mode for detailed tracing}
define the-works = 3 { verbose, plus check of postamble if random-reading }
```

⟨ Globals in the outer block 10 ⟩ +≡

```
out-mode: errors-only .. the-works; { controls the amount of output }
max-pages: integer; { at most this many bop .. eop pages will be printed }
resolution: real; { pixels per inch }
new-mug: integer; { if positive, overrides the postamble's magnification}
```

42. The starting page specification is recorded in two global arrays called ***start-count*** and ***start-there.*** For example, '1.*.-5' is represented by ***start-there [0] = true, start-count [0]*** = 1, *start_there*[1] = *false*, *start_there*[2] = ***true, start-count*** [2] = ***-5.*** We also set ***start-vals*** = 2, to indicate that count 2 was the last one mentioned. The other values of ***start-count*** and ***start-there*** are not important, in this example.

⟨ Globals in the outer block 10 ⟩ +≡

```
start-count: array [0 .. 9] of integer; { count values to select starting page }
start-there: array [0 .. 9] of boolean; { is the start-count value relevant? }
start-vals: 0 .. 9; { the last count considered significant }
count: array [0 .. 9] of integer; { the count values on the current page }
```

43. ⟨ Set initial values 11 ⟩ +≡

```
out-mode ← the-works; max-pages ← 1000000; start-vals ← 0; start_there[0] ← false;
```

44. Here is a simple subroutine that tests if the current page might be the starting page.

```
function start-match: boolean; { does count match the starting spec? }
  var k: 0..9; { loop index }
    match,: boolean,; { does everything match so far? }
  begin match ← true;
  for k ← 0 to start-vals do
    if sturt-there[k] A (start-count[k] ≠ count [k]) then match ← false;
  start-match ← match;
  end:
```

45. The ***input-Zn*** routine waits for the user to type a line at his or her terminal; then it puts ASCII-code equivalents for the characters on that line into the *buffer* array. The ***term-in*** file is used for terminal input. and ***term-out*** for terminal output.

```
⟨ Globals in the outer block 10 ⟩+≡
buffer : array [0 .. terminal_line_length] of ASCII-code;
term-in: text-file;   { the terminal, considered as an input file }
term-out: text-file;   { the terminal, considered as an output file }
```

46. Since the terminal is being used for both input and output, some systems need a special routine to make sure that the user can see a prompt message before waiting for input based on that message. (Otherwise the message may just be sitting in a hidden buffer somewhere, and the user will have no idea what the program is waiting for.) We shall invoke a system-dependent subroutine ***update-terminal*** in order to avoid this problem.

```
define update-terminal ≡ break( term-out)   { empty the terminal output buffer }
```

47. During the dialog, `DVItype` will treat the first blank space in a line as the end of that line. Therefore ***input-Zn*** makes sure that there is always at least one blank space in *buffer*.

```
procedure input_ln;   { inputs a line from the terminal }
  var k: 0.. terminal-line-length ;
  begin update_terminal; reset (term-in);
  if eoln (term-in) then read-ln (term-in);
  k ← 0;
  while (k < terminal-line-length) A ¬eoln( term-in) do
    begin buffer [k] ← xord [ term-in ↑]; incr (k); get (term-in);
    end;
  buffer[k] + "␣";
  end;
```

48. The global variable ***buf-ptr*** is used while scanning each line of input; it points to the first unread character in *buffer*.

```
⟨ Globals in the outer block 10 ⟩+≡
buf-ptr : 0 .. terminal-line-length;   { the number of characters read }
```

49. Here is a routine that scans a (possibly signed) integer and computes the decimal value. If no decimal integer starts at ***buf-ptr.*** the value 0 is returned. The integer should be less than 2^{31} in absolute value.

```
function get-integer: integer;
  var x: integer;   { accumulates the value }
    negative : boolean;   { should the value be negated? }
  begin if buffer[buf_ptr] = "-" then
    begin negative ← true: incr( buf-ptr);
    end
  else negative ← false;
  x ← 0;
  while (buffer[ buf-ptr] ≥ "0") A (buffer [buf-ptr] ≤ "9") do
    begin x + 10 * x + buffer[buf_ptr] − "0": incr( buf-ptr);
    end;
  if negative then get-integer + -x else get-integer + x;
  end;
```

50. The selected options are put into global variables by the ***dialog*** procedure, which is called just as DVItype begins.

```
procedure dialog;
  label 1, 2, 3, 4, 5;
  var k: integer;  { loop variable }
  begin rewrite( term-out);  { prepare the terminal for output }
  write-Zn( term-out, banner); ( Determine the desired out-mode 51 );
  ( Determine the desired start-count values 52);
  ( Determine the desired max-pages 53);
  ( Determine the desired resolution 54);
  ( Determine the desired new-mug 55);
  (Print all the selected options 56);
  end;
```

51. (Determine the desired ***out-mode*** 51) ≡

```
1: write (term-out, 'Output level (default=3, ? for help) : '); out-mode ← the-works; input-Zn;
  if buffer[0] ≠ " " then
    if (buffer[0] ≥ "0") A (buffer[0] ≤ "3") then out-mode ← buffer[0] - "0"
    else begin write (term-out, 'Type 3 for complete listing, ');
      write(term_out, ' 0 for errors only, ');
      write-Zn (term-out, ' 1 or 2 for something in between. '); goto 1;
      end
```

This code is used in section 50.

52. (Determine the desired ***start-count*** values 52) ≡

```
2: write (term-out, 'Starting page (default=*) : '); start_vals ← 0: start_there[0] ← false; input-Zn;
  buf-ptr ← 0; k ← 0;
  if buffer[0] ≠ " " then
    repeat if buffer[buf_ptr] = "*" then
        begin start_there[k] ← false; incr( buf_ptr);
        end
      else begin start_there[k] ← true: start_count[k] ← get-integer;
        end:
      if (k < 9) ∧ (buffer[buf_ptr] = ".") then
        begin incr (k); incr(buf_ptr);
        end
      else if buffer[buf_ptr] = " " then start_vals ← k
        else begin write(term_out, 'Type, e.g., 1.*.-5 to specify the ');
          write_ln(term_out, 'first page with \count0=1, \count2=-5. '); goto 2:
          end:
    until start_vals = k
```

This code is used in section 50.

53. ⟨ Determine the desired *max_pages* 53 ⟩≡

```
3: write (term-out, 'Maximum␣number␣of␣pages␣(default=1000000): ␣'); mux-pages ← 1000000;
   input-Zn; buf-ptr ← 0;
   if buffer[0] ≠ "␣" then
     begin mux-pages ← get-integer;
     if mux-pages ≤ 0 then
       begin write-Zn( term-out, 'Please␣type␣a␣positive␣number.'); goto 3;
       end;
     end
```

This code is used in section 50.

54. ⟨ Determine the desired ***resolution*** 54 ⟩≡

```
4: write (term-out, 'Assumed␣device␣resolution');
   write (term-out, '␣in␣pixels␣per␣inch␣(def ault=300/1): ␣'); resolution ← 300.0; input_ln;
   buf-ptr ← 0;
   if buffer[0] ≠ "␣" then
     begin k ← get-integer;
     if (k > 0) A (buffer[buf_ptr] = "/") A (buffer[buf_ptr + 1] > "0") A (buffer[buf_ptr + 1] ≤ "9") then
       begin incr (buf_ptr); resolution ← k/get-integer;
       end
     else begin write (term-out, 'Type␣a␣ratio␣of␣positive␣integers;');
       write-Zn ( term-out, '␣(1␣pixel␣per␣mm␣would␣be␣254/10).'); goto 4;
       end;
     end
```

This code is used in section 50.

55. ⟨ Determine the desired ***new-mug*** 55 ⟩≡

```
5: write (term-out, 'New␣magnification␣(default=0␣to␣keep␣the␣old␣one):␣'); new-mug ← 0;
   input-Zn; buf-ptr ← 0;
   if buffer[0] ≠ "␣" then
     if (buffer[0] ≥ "0") A (buffer[0] ≤ "9") then new-mug ← get-integer
     else begin write (term-out, 'Type␣a␣positive␣integer␣to␣override␣');
       write-Zn (term-out , 'the␣magnif ication␣in␣the␣DVI␣f ile.'); goto 5;
       end
```

This code is used in section 50.

56. After the dialog is over, we print the options so that the user can see what **DVItype** thought was specified.

```
⟨ Print all the selected options 56⟩ ≡
  print-Zn('Options␣selected:'); print('␣␣Starting␣page␣=␣');
  for k ← 0 to start_vals do
    begin if start_there[k] then print(start_count[k] : 1)
    else print ('*');
    if k < start_vals then print ('.')
    else print-Zn ('␣');
    end;
  print-Zn ('␣␣Maximum␣number␣of␣pages␣=␣', mox-pages : 1);
  print ('␣␣Output␣level␣=␣', out-mode : 1);
  case out-mode of
  errors_only: print-Zn('␣(showing␣bops,␣fonts,␣and␣error␣messages␣only)');
  terse: print-Zn('␣(terse)');
  verbose: print-Zn ('␣(verbose)');
  the-works: if random-reading then print-Zn('␣(the␣works)')
    else begin out-mode ← verbose; print-Zn ('␣(the␣works:␣same␣as␣level␣2␣in␣this␣DVItype)');
      end;
  end;
  print-Zn('␣␣Resolution␣=␣', resolution : 12 : 8, '␣pixels␣per␣inch');
  if new-mug > 0 then print-Zn('␣␣New␣magnification␣factor␣=␣', new_mag/1000 : 8 : 3)
```

This code is used in section 50.

57. Defining fonts. When ***out-mode*** = ***the-works,*** DVItype reads the postamble first and loads all of the fonts defined there: then it processes the pages. In this case, a *fnt_def* command should match a previous definition if and only if the ***jnt-dej*** being processed is not in the postamble. But if ***out-mode*** < ***the-works,*** DVItype reads the pages first and the postamble last, so the conventions are reversed: a ***jnt-dej*** should match a previous ***jnt-dej*** if and only if the current one is a part of the postamble.

A global variable *in_postamble* is provided to tell whether we are processing the postamble or not.

```
⟨ Globals in the outer block 10 ⟩ +≡
in-postumble : boolean;  { are we reading the postamble? }
```

58. ⟨ Set initial values 11 ⟩ +≡

```
in_postamble ← false;
```

59. The following subroutine does the necessary things when a ***jnt-dej*** command is being processed.

```
procedure define-font (e : integer):  { e is an external font number }
  var f: 0 .. mux-fonts; p: integer;  { length, of the area/directory spec }
    n: integer:  { length of the font name proper}
    c, q, d: integer:  { check sum, scaled size, and design size }
    r: 0 .. name-length;  { index into cur-name }
    j, k: 0 .. name-size;  { indices into names }
    mismatch : boolean ;  { do names disagree? }
  begin if nf = max_fonts then
    abort('DVItype capacity exceeded (max fonts=', mux-fonts : 1, ')!');
  font-num [nj] ← e; j ← 0;
  while font_num[f] ≠ e do incr(f);
  ⟨ Read the font parameters into position for font nf, and print the font name 61 ⟩;
  if ((out-mode = the-works) ∧ in-postumble) ∨ ((out-mode < the-works) ∧ ¬in_postamble) then
    begin if f < nj then print-Zn('---this font was already defined!');
    end
  else begin if f = nf then print-Zn ('---this font wasn't loaded before!');
    end;
  if f = nf then ⟨ Load the new font. unless there are problems 62 ⟩
  else ⟨ Check that the current font definition matches the old one 60 ⟩;
  end;
```

60. ⟨ Check that the current font definition matches the old one 60 ⟩ ≡

```
  begin if font-check-sum[j] ≠ c then
    print-Zn ('---check sum doesn't match previous def inition!');
  if font_scaled_size[f] ≠ q then print-Zn ('---scaled size doesn't match previous definition!');
  if font-design-site[j] ≠ d then print-Zn ('---design size doesn't match previous definition!');
  j ← font_name[f]; k ← font_name[nf]; mismatch ← false;
  while j < font_name[f + 1] do
    begin if names[j] ≠ names[k] then mismatch ← true;
    incr(j); incr (k);
    end;
  if k ≠ font_name[nf + 1] then mismatch ← true;
  if mismatch then print-Zn ('---f ont name doesn't match previous definition!');
  end
```

This code is used in section 59.

61. ⟨ Read the font parameters into position for font *nf*, and print the font name 61) ≡
```
c ← signed-quad; font-check-sum[ nj] ← c;
q ← signed-quad: font-scaled-size [nj ] ← q;
d ← signed-quad; font-design-size [nj ] ← d;
p ← get-byte; n ← get-byte;
if font_name [ nj ] + n + p > name-size then
  abort('DVItype capacity exceeded (name size=', name-size : 1, ')!');
font-name [nf + 1] ← font-name [nj ] + n + p;
if showing then print (': ') { when showing is true, the font number has already been printed }
else print('Font ', e : 1, ': ');
if n + p = 0 then print ('null font name!')
else for k ← font_name[nf] to font_name[nf + 1] − 1 do names[k] ← get-byte;
incr( nj); print-font (nj − 1); decr( nf )
```
This code is used in section 59.

62. (Load the new font, unless there are problems 62 ⟩ ≡
```
begin ( Move font name into the cur-name string 66 );
open_tfm_file;
if eoj ( tfm_file ) then print ('---not loaded, TFM file can''t be opened!')
else begin if (q ≤ 0) ∨ (q ≥ '1000000000) then print ('---not loaded, bad scale (', q : 1, ')!')
  else if (d ≤ 0) ∨ (d ≥ '1000000000) then print('---not loaded, bad design size (', d : 1, ')!')
    else if in_TFM (q) then ( Finish loading the new font info 63 );
  end;
if out-mode = errors_only then print_ln(' ');
end
```
This code is used in section 59.

63. (Finish loading the new font info 63 ⟩ ≡
```
begin font_space[nf] ← q div 6; { this is a 3-unit "thin space" }
if (c ≠ 0) A (tjm-check-sum ≠ 0) A (c ≠ tfm_check_sum) then
  begin print_ln('---beware: check sums do not agree!');
  print_ln('   (', c : 1, ' vs. ', tfm_check_sum : 1, ')'); print('   ');
  end;
print('---loaded at size ', q : 1, ' DVI units'); d ← round((100.0 * conv * q)/(true_conv * d));
if d ≠ 100 then
  begin print_ln(' '); print(' (this font is magnified ', d : 1, '%)');
  end;
incr( nj ): {now the new font is officially present }
font_space [nj] ← 0; { for out_space and out_vmove }
end
```
This code is used in section 62.

64. If *p* = 0, i.e., if no font directory has been specified. DVItype is supposed to use the default font directory. which is a system-dependent place where the standard fonts are kept. The string variable *default_directory* contains the name of this area.
```
define default_directory_name ≡ 'TeXf onts : '  { change this to the correct name }
define default_directory_name_length = 9  { change this to the correct, length }
```
(Globals in the outer block 10 ⟩ +≡
```
default_directory: packed array [1 .. default_directory_name_length] of char;
```

65. ⟨ Set initial values 11) +≡
```
default-directory + default-directory-name;
```

66. The string *cur_name* is supposed to be set to the external name of the TFM file for the current font. This usually means that we need to prepend the name of the default directory, and to append the suffix '.TFM'. Furthermore, we change lower case letters to upper *case,* since *cur_name* is a Pascal string.

⟨ Move font name into the *cur-name* string 66 ⟩ ≡
```
for k ← 1 to name-length do cur-name [k] ← ' ';
if p = 0 then
  begin for k ← 1 to default-directory-name-length do cur-name [k] ← default-directory [k];
  r ← default-directory_name-length;
  end
else r ← 0:
for k ← font-name [nj] to font-name [nj + 1] − 1 do
  begin incr (r);
  if r + 4 > name-length then
    abort('DVItype capacity exceeded (max font name length=', name-length : 1, ')!');
  if (names [k] ≥ "a") A (names[k] ≤ "z") then cur_name[r] ← xchr[names[k] − '40]
  else cur-name [r] ← xchr[names[k]];
  end;
cur-name [r + 1] ← '.'; cur-name [r + 2] ← 'T'; cur-name [r + 3] + 'F'; cur_name[r + 4] + 'M'
```
This code is used in section 62.

67. Low level output routines. Simple text in the DVI file is saved in a buffer until ***line-length*** − 2 characters have accumulated, or until some non-simple DVI operation occurs. Then the accumulated text is printed on a line, surrounded by brackets. The global variable ***text-ptr*** keeps track of the number of characters currently in the buffer.

```
( Globals in the outer block 10) +≡
text-ptr: 0 . . line-length;   {the number of characters in text-buj }
text-buj: array [1 . . line-length] of ASCII-code;   { saved characters }
```

68. (Set initial values 11 ⟩ +≡

```
  text-ptr ← 0;
```

69. The ***flush-text*** procedure will empty the buffer if there is something in it.

```
procedure flush-text;
  var k: 0 . . line-length;   { index into text-buj }
  begin if text-ptr > 0 then
    begin if out-mode > errors-only then
      begin print ('[');
      for k ← 1 to text-ptr do print(xchr[text_buf[k]]);
      print_ln(']');
      end;
    text-ptr ← 0;
    end;
  end;
```

70. And the ***out-text*** procedure puts something in it.

```
procedure out-text (c : ASCII-code);
  begin if text-ptr = line-length − 2 then flush-text;
  incr (text-ptr); text_buf[text_ptr] ← c;
  end;
```

71. Translation to symbolic form. The main work of DVItype is accomplished by the ***do-page*** procedure, which produces the output for an entire page, assuming that the ***bop*** command for that page has already been processed. This procedure is essentially an interpretive routine that reads and acts on the DVI commands.

72. The definition of DVI files refers to six registers, ***(h,*** v, w, x, y, x), which hold integer values in DVI units. In practice, we also need registers ***hh*** and vv, the pixel analogs of ***h*** and v, since it is not always true that ***hh = pixel-round(h)*** or vv ***= pixel-round(v).***

The stack of ***(h,*** v, w, x, y, z) values is represented by eight arrays called ***hstack, . . .,*** $zstack$, ***hhstack,*** and ***vvstack.***

⟨ Globals in the outer block 10 ⟩ +≡
h. v, w. x, y, z, ***hh,*** vv: ***integer;*** { current state values }
hstack, vstack, $wstack$, $xstack$, $ystack$, $zstack$: **array [0 . .** ***stack-size]*** **of** ***integer;***
{ pushed down values in DVI units }
hhstack, vvstack: **array** [0 . . ***stack-size]*** **of** ***integer;*** { pushed down values in pixels }

73. Three characteristics of the pages (their max-v, ***max-h,*** and ***max-s)*** are specified in the postamble, and a warning message is printed if these limits are exceeded. Actually ***max-v*** is set to the maximum height plus depth of a page, and max_h to the maximum width, for purposes of page layout. Since characters can legally be set outside of the page boundaries, it is not an error when max_v or ***max-h*** is exceeded. But ***max-s*** should not be exceeded.

The postamble also specifies the total number of pages; DVItype checks to see if this total is accurate.

(Globals in the outer block 10 ⟩ +≡
max-v: integer; { the value of $abs(v)$ should probably not exceed this }
max-h: integer; { the value of ***abs*** (h) should probably not exceed this }
max-s: integer; { the stack depth should not exceed this }
max-v-so-jar, $max\_h\_so\_far$***, max-s-so-fur:*** ***integer;*** { the record high levels }
total-pages: integer; { the stated total number of pages }
page-count : ***integer;*** { the total number of pages seen so far }

74. (Set initial values 11) +≡
max-v ← ***'17777777777 − 99; max-h*** ← ***'17777777777 − 99; max-s*** ← ***stack-size*** + 1;
$max\_v\_so\_far$ ← ***0;*** $max\_h\_so\_far$ ← ***0:*** $max\_s\_so\_far$ ← ***0: page-count*** ← ***0;***

75. Before we get into the details of &o-page, it is convenient to consider a simpler routine that computes the first parameter of each opcode.

```
define four_cases(#) ≡ #, # + 1, # + 2, # + 3
define eight-cases(#) ≡ four-cases(#), four_cases(# + 4)
define sixteen-cases(#) ≡ eight-cases(#), eight-cases(# + 8)
define thirty-two-cases(#) ≡ sixteen-cases(#), sixteen-cases(# + 16)
define sixty-four-cases(#) ≡ thirty-two-cases(#), thirty-two-cases(# + 32)
function first-par(o : eight-bits): integer;
  begin case o of
  sixty-four-cases (set-char-0), sixty-four-cases (set-char-0 + 64): first-par ← o − set-char-0;
  set1, put1, fntl, xxx1, fnt-defl : first-pur ← get-byte;
  set1 + 1, put1 + 1, fntl + 1, xxx1 + 1, fnt-defl + 1: first-par ← get-two-bytes;
  set1 + 2, put1 + 2, fntl + 2, xxx1 + 2, fnt-defl + 2: first-par ← get-three-bytes;
  right1, wl, xl, down1, y1, z1: first-par ← signed-byte;
  right1 + 1, wl + 1, xl + 1, down.1 + 1, y1 + 1, z1 + 1: first-par ← signed-pair;
  right1 + 2, wl + 2, xl + 2, down1 + 2, yl + 2, z1 + 2: first-par ← signed-trio;
  set1 + 3, set-rule, put1 + 3, put-rule, right1 + 3, wl + 3, xl + 3, down1 + 3, yl + 3, z1 + 3, fntl + 3,
      xxx1 + 3, fnt-defl + 3: first-par ← signed-quad;
  nop, bop, eop, push, pop, pre, post, post-post, undefined-commands: first-par ← 0;
  w0: first-par ← w;
  x0 : first-par ← x;
  y0: first-pur ← y;
  z0: first-pur ← z;
  sixty-four-cuses(fnt-num-0 ): first-par ← 0 − fnt_num_0;
  end;
  end;
```

76. Here is another subroutine that we need: It computes the number of pixels in the height or width of a rule. Characters and rules will line up properly if the sizes are computed precisely as specified here. (Since *conv* is computed with some floating-point roundoff error, in a machine-dependent way, format designers who are tailoring something for a particular resolution should not plan their measurements to come out to an exact integer number of pixels; they should compute things so that the rule dimensions are a little less than an integer number of pixels. e.g., 4.99 instead of 5.00.)

```
function rule_pixels(x : integer): integer:  { computes ⌈conv . x⌉ }
  var n: integer; .
  begin n ← trunc( conv∗x);
  if n < conv ∗ x then rule-pixels ← n + 1 else rule-pixels ← n;
  end:
```

77. Strictly speaking, the ***do-page*** procedure is really a function with side effects. not a **'procedure'**; it returns the value *false* if `DVItype` should be aborted because of some unusual happening. The subroutine is organized as a typical interpreter, with a multiway branch on the command code followed by **goto** statements leading to routines that finish up the activities common to different commands. We will use the following labels:

```
define fin-set = 41   { label for commands that set or put a character }
define fin-rule = 42   { label for commands that set or put a rule }
define move-right = 43   { label for commands that change h }
define move-down = 44   { label for commands that change v }
define show-stute = 45   { label for commands that change s }
define change_font = 46   { label for commands that change cur-font }
```

78. Some Pascal compilers severely restrict the length of procedure bodies, so we shall split ***do-page*** into two parts, one of which is called ***special-cases.*** The different parts communicate with each other via the global variables mentioned above, together with the following ones:

```
⟨ Globals in the outer block 10 ⟩ +≡
s: integer;  { current stack size }
ss: integer;  { stack size to print }
cur-font: integer;  { current internal font number }
showing : boolean ;  { is the current command being translated in full? }
```

79. Here is the overall setup.

```
⟨ Declare the function called special-cases 82 ⟩
function do-page: boolean;
  label fin-set, fin_rule, move-right, show-state, done, 9998, 9999;
  var o: eight-bits;  { operation code of the current command }
    p, q: integer;  { parameters of the current command }
    a: integer;  (byte number of the current command }
    hhh : integer:  { h, rounded to the nearest pixel }
  begin cur-font ← nf;  { set current font undefined }
  s ← 0; h ← 0; v ← 0; w ← 0; x ← 0; y ← 0; z ← 0; hh ← 0; vv ← 0;  { initialize the state variables }
  while true do ⟨ Translate the next command in the DVI file; goto 9999 with do-page = true if it was
      eop; goto 9998 if premature termination is needed 80);
9998: print_ln ( ' ! ' ): do-page ← false;
9999: end;
```

80. Commands are broken down into "major" and "minor" categories: A major command is always shown in full, while a minor one is put into the buffer in abbreviated form. Minor commands, which account for the bulk of most DVI files, involve horizontal spacing and the typesetting of characters in a line; these are shown in full only if ***out-mode*** $\geq$ ***verbose.***

```
define show(#) ≡
        begin flush_text; showing ← true; print(a : 1, ´:␣´, #);
        end
define major (#) ≡
        if out-mode > errors-only then show (#)
define minor (#) ≡
        if out-mode ≥ verbose then
          begin showing ← true; print(a : 1, ´ :␣´, #);
          end
define error(#) ≡
        if ¬showing then show(#)
        else print (´␣´, #)
```

(Translate the next command in the DVI file; **goto** 9999 with ***do-page*** = ***true*** if it was ***eop;*** **goto 9998** if premature termination is needed 80 ⟩≡

```
begin a ← cur-lot; showing ← false; o ← get-byte; p ← first-par(o);
if eof (dvi-file) then bad-dvi (´the␣file␣ended␣prematurely´);
( Start translation of command o and goto the appropriate label to finish the job 81);
```

fin-set: (Finish a command that either sets or puts a character, then **goto** ***move-right*** or ***done*** 89);
fin-rule: (Finish a command that either sets or puts a rule, then **goto** ***move-right*** or ***done*** 90 ⟩;
move-right: (Finish a command that sets $h \leftarrow h + q$, then **goto** ***done*** 91 ⟩;
show-state: (Show the values of ss, h, v, w, x, y, z, ***hh,*** and vv; then **goto** ***done*** 93 ⟩;

```
done: if showing then print-ln( ´␣´);
  end
```

This code is used in section 79.

81. The multiway switch in *first-par*, above, was organized by the length of each command; the one in ***do-page*** is organized by the semantics.

(Start translation of command o and **goto** the appropriate label to finish the job 81) ≡

```
if o < set-char-0 + 128 then ( Translate a set-char command 88 ⟩
else case 0 of
  four_cases(set1): begin major( 'set ´, o − set1 + 1 : 1, ´␣´, p : 1); goto fin-set;
    end;
  four_cases(put1): begin major( 'put ´, o − put1 + 1 : 1, ´␣´, p : 1); goto fin-set;
    end;
  set-rule: begin major(´setrule´); goto fin_rule;
    end:
  put-rule: begin major (´putrule´); goto fin-rule;
    end;
  ( Cases for commands nop, bop, . . . , pop 83)
  ( Cases for horizontal motion 84)
  othercases if special-cases (o, p, a) then goto done else goto 9998
  endcases
```

This code is used in section 80.

82. (Declare the function called ***special-cases.*** 82) ≡

```
function special_cases(o : eight-bits; p, a : integer): boolean;
  label change-font, move-down, done, 9998;
  var q: integer;  { parameter of the current command }
    k: integer;  { loop index }
    bad-char: boolean;  { has a non-ASCII character code appeared in this xxx? }
    pure : boolean ;  { is the command error-free? }
    vvv : integer;  { v, rounded to the nearest pixel }
  begin pure ← true;
  case 0 of
  ( Cases for vertical motion 85 )
  ( Cases for fonts 86 )
  four_cases(xxx1 ): ( Translate an xxx command and goto done 87);
  pre: begin error('preamble␣command␣within␣a␣page!'); goto 9998;
    end;
  post, post-post: begin error ('postamble␣command␣within␣a␣page!'); goto 9998:
    end;
  othercases begin error('undefined␣command␣'.o : 1.'!'); goto done;
    end
  endcases;
move-down: ( Finish a command that sets v ← v + p, then goto done 92 );
change-font: ( Finish a command that changes the current font, then goto done 94 );
9998: pure ← false;
done: special-cases ← pure:
  end;
```

This code is used in section 79.

83. ⟨ Cases for commands ***nop, bop, ... , pop*** 83 ⟩≡

```
nop: begin minor( 'nop '); goto done;
  end;
bop: begin error ('bop␣occurred␣bef ore␣eop ! '); goto 9998;
  end;
eop: begin major('eop');
  if s ≠ 0 then error('stack␣not␣empty␣at␣end␣of␣page␣(level␣',s : 1,')! ');
  do-page ← true; print-Zn('␣'); goto 9999;
  end;
push : begin major ( 'push ');
  if s = max-s-so-far then
    begin max_s_so_far ← s + 1;
    if s = max_s then error('deeper␣than␣claimed␣in␣postamble ! ');
    if s = stack-size then
      begin error('DVItype␣capacity␣exceeded␣(stack␣size=',stack-size : 1,')');goto 9998;
      end;
    end;
  hstack[s] ← h; vstack [s] ← v; wstack [s] ← w; xstack[s] t x; ystack[s] t y; zstack[s] t z;
  hhstack[s] ← hh; vvstack[s] ← vv; incr(s); ss t s − 1; goto show-state;
  end;
pop : begin major ('pop');
  if s = 0 then error('(illegal␣at␣level␣zero)! ')
  else begin decr(s); hh ← hhstack[s]; vv t vvstack[s]; h t hstack[s]; v t vstack[s]; w t wstack[s];
    x ← xstack [s]; y ← ystack[s]; z ← zstack[s];
    end;
  ss ← s; goto show_state;
  end;
```

This code is used in section 81.

84. Rounding to the nearest pixel is best done in the manner shown here, so as to be inoffensive to the eye: When the horizontal motion is small, like a kern, ***hh*** changes by rounding the kern; but when the motion is large. ***hh*** changes by rounding the true position ***h*** so that accumulated rounding errors disappear. We allow a larger space in the negative direction than in the positive one. because TEX makes comparatively large backspaces when it positions accents.

```
define out-space (#) ≡
        if (p ≥ font_space [cur-font]) ∨ (p ≤ -4 * font_space[cur_font]) then
          begin out-text ("␣"); hh ← pixel-round (h + p);
          end
        else hh ← hh + pixel-round(p);
      minor(#, '␣',p : 1); q ← p; goto move-righf
```

(Cases for horizontal motion 84) ≡

```
four-cases (right1 ): begin out_space( 'right ', o − right1 + 1 : 1):
  end;
w0 , four_cases( w1 ): begin w ← p: out_space('w', o − w0 : 1):
  end;
x0 , four_cases( x1 ): begin x ← p; out-spare ('x', o − x0 : 1):
  end;
```

This code is used in section 81.

85. Vertical motion is done similarly, but with the threshold between "small" and "large" increased by a factor of five. The idea is to make fractions like "$\frac{1}{2}$" round consistently, but to absorb accumulated rounding errors in the baseline-skip moves.

```
define out-vmove (#) ≡
        if abs(p) ≥ 5 * font_space[cur_font] then vv ← pixel_round(v + p)
        else vv ← vv + pixel-round(p);
      major (#, ´␣´, p : 1); goto move-down
```

(Cases for vertical motion 85) ≡

```
four_cases( down1 ): begin out-vmove( 'down ? o − down1 + 1 : 1);
  end;
y0 , four-cases (y1 ): begin y ← p; out_vmove( ´y´, o − y0 : 1);
  end;
z0 , four_cases( z1 ): begin z ← p; out_vmove( ´z´, o − z0 : 1);
  end:
```

This code is used in section 82.

86. (Cases for fonts 86) ≡

```
sixty_four_cases(fnt_num_0 ): begin major( ´fntnum´, p : 1); goto change-font;
  end;
four-cases (fnt1 ): begin major( fnt ´, o − fnt1 + 1 : 1, ´␣´, p : 1); goto change-font;
  end;
four-cases (fnt_def1 ): begin major ( ´f ntdef ´, o − fnt_def1 + 1 : 1, ´␣´, p : 1); define-font(p); goto done;
  end;
```

This code is used in section 82.

87. (Translate an xxx command and goto *done* 87) ≡

```
begin major (´xxx␣´´´); bad-char ← false;
if p < 0 then error( ´string␣of␣negative␣length!´);
for k ← 1 to p do
  begin q ← get-byte;
  if (q < "␣") V (q > "~") then bad-char ← true;
  if showing then print (xchr [q]);
  end;
if showing then print (´´´´);
if bad-char then error( ´non-ASCII␣character␣in␣xxx␣command!´);
goto done;
end
```

This code is used in section 82.

88. ⟨ Translate a *set-char* command 88 ⟩ ≡

```
begin if (o > "␣") A (o ≤ "~") then
  begin out-text(p); minor( ´setchar´, p : 1);
  end
else major( ´setchar´, p : 1);
goto fin-set;
end
```

This code is used in section 81.

89. (Finish a command that either sets or puts a character, then **goto** move-right or *done* 89 ⟩ ≡

```
if p < 0 then p ← 255 − ((-1 -p) mod256)
else if p ≥ 256 then p ← p mod 256;  { width computation for oriental fonts }
if (p < font_bc[cur_font]) ∨ (p > font_ec[cur_font]) then q ← invalid-width
else q ← char-width (cur-font)(p);
if q = invalid-width then
  begin error('character␣', p : 1, '␣invalid␣in␣font␣'); print-font (cur-font);
  if cur-font ≠ nj then print ('!');  { font nj has '!' in its name }
  end;
if o ≥ put1 then goto done;
if q = invalid-width then q ← 0
else hh ← hh + char-pixel-uridth (cur-font)(p);
goto move-right
```

This code is used in section 80.

90. (Finish a command that either sets or puts a rule, then **goto** *move-right* or *done* 90 ⟩ ≡

```
q ← signed-quad;
if showing then
  begin print('␣height␣', p : 1, '␣width␣', q : 1);
  if (p ≤ 0) ∨ (q ≤ 0) then print ('␣(invisible)')
  else print ('␣(', rule-pixels(p) : 1, 'x', rule-pixels(q) : 1, '␣pixels)');
  end;
if o = put-rule then goto done;
if showing then print-ln('␣');
hh ← hh + rule-pixels(q); goto move-right
```

This code is used in section 80.

91. A sequence of consecutive rules, or consecutive characters in a fixed-width font whose width is not an integer number of pixels, can cause **hh** to drift far away from a correctly rounded value. DVItype ensures that the amount of drift will never exceed *max_drift* pixels.

Since DVItype is intended to diagnose strange errors, it checks carefully to make sure that **h** and v do not get out. of range. Normal DVI-reading programs need not do this.

```
define infinity ≡ 17777777777  { ∞ (approximately) }
define max_drift = 2  { we insist that abs( hh − pixel-round(h)) ≤ max_drift }
⟨ Finish a command that sets h ← h + q, then goto done 91 ⟩ ≡
  if (h > 0) A (q > 0) then
    if h > infinity − q then
      begin error('arithmetic␣overflow!␣parameter␣changed␣from␣', q : 1, '␣to␣', infinity − h : 1);
      q ← infinity − h;
      end;
  if (h < 0) A (q < 0) then
    if -h > q + infinity then
      begin error('arithmetic␣overflow!␣parameter␣changed␣from␣', q : 1, '␣to␣', (−h) − infinity : 1);
      q ← (−h) − infinity;
      end;
  hhh ← pixel-round (h + q):
  if abs(hhh − hh) > max_drift then
    if hhh > hh then hh ← hhh − max_drift
    else hh ← hhh + max_drift;
  if showing then
    begin print('␣h:=', h : 1);
    if q ≥ 0 then print ('+');
    print(q : 1, '=', h + q : 1. '␣hh:=', hh : 1):
    end:
  h ← h + q;
  if abs(h) > max_h_so_far then
    begin if abs(h) > max-h + 99 then
      begin error('warning:␣|h|>', max-h : 1, '!'); max-h ← abs(h);
      end;
    max-h-so-jar ← abs (h);
    end:
  goto done
```

This code is used in section 80.

92. ⟨ Finish a command that sets $v \leftarrow v + p$, then **goto done** 92 ⟩ ≡

```
if (v > 0) A (p > 0) then
  if v > infinity − p then
    begin error('arithmetic␣overflow!␣parameter␣changed␣from␣', p : 1, '␣to␣', infinity − v : 1);
    p ← infinity − v;
    end:
if (v < 0) A (p < 0) then
  if −v > p + infinity then
    begin error('arithmetic␣overflow!␣parameter␣changed␣from␣', p : 1, '␣to␣', (−v) − infinity : 1);
    p ← (-v) − infinity;
    end;
vvv ← pixel-round (v + p);
if abs( vvv − vv) > max_drift then
  if vvv > vv then vv ← vvv − max_drift
  else vv ← vvv + max_drift;
if showing then
  begin print('␣v:=', v : 1);
  if p ≥ 0 then print('+');
  print(p : 1, '=', v + p : 1, ',␣vv:=', vv : 1);
  end;
v ← v 4-p;
if abs(v) > max_v_so_far then
  begin if abs (v) > mnx-v + 99 then
    begin error('warning:␣|v|>', max_v : 1, '!'); max_v ← abs(v);
    end:
  max_v_so_far ← abs(v);
  end;
goto done
```

This code is used in section 82.

93. ⟨ Show the values of ss, h, v, w, x, y, z, **hh,** and vv; then **goto** *done* 93 ⟩ ≡

```
if showing then
  begin print-ln('␣'); print( level,': ss : 1, ': (h=', h : 1, ',v=', v : 1, ',w=', w : 1, ',x=', x : 1, ',y=',
      y : 1. ',z=', z : 1. ',hh=', hh : 1. ',vv=', vv : 1, ')');
  end:
goto done
```

This code is used in section 80.

94. ⟨ Finish a connnand that changes the current font, then **goto** *done* 94 ⟩ ≡

```
font_num[ nf ] ← p: cur-font ← 0:
while font-num [ cur-font] ≠ p do incr (cur-font):
if showing then
  begin print ( '␣current␣f ont␣is␣'); print-font (cur-font):
  end:
goto done
```

This code is used in section 82.

95. Skipping pages. A routine that's much simpler than ***do-page*** is used to pass over pages that are not being translated. The ***skip-pages*** subroutine is assumed to begin just after the preamble has been read, or just after a ***bop*** has been processed. It continues until either finding a ***bop*** that matches the desired starting page specifications, or until running into the post amble.

```
procedure skip-pages ;
  label 9999:  { end of this subroutine }
  var p: integer;  { a parameter }
    k: 0 . . 255;  { command code }
    down-the-drain: integer; { garbage }
  begin showing ← false;
  while true do
    begin if eof(dvi_file) then bad_dvi('the␣file␣ended␣prematurely');
    k ← get-byte; p ← first-par(k);
    case k of
    bop: begin (Pass a bop command, setting up the count array 98);
      if ¬started ∧ start-match then
        begin started ← true; got0 9999;
        end;
      end;
    set-rule, put-rule: down-the-drain ← signed-quad;
    fnt-defl , fnt-defl + 1, fnt-defl + 2, fnt-defl + 3: begin define-font(p); print-ln('␣');
      end;
    xxx1 , xxx1 + 1, xxx1 + 2, xxx1 + 3: while p > 0 do
        begin down-the-drain ← get-byte; decr(p);
        end;
    post: begin in-postamble ← true; goto 9999;
      end;
    othercases do-nothing
    endcases;
    end:
9999: end;
```

96. Global variables called *old_backpointer* and ***new,backpointer*** are used to check whether the back pointers are properly set up. Another one tells whether we have already found the starting page.

```
( Globals in the outer block 10 ⟩ +≡
old-backpointer : integer;  { the previous bop command location }
new-backpointer: integer;  { the current bop command location }
started: boolean;  { has the starting page been found? }
```

97.

```
( Set initial values 11) +≡
  old_backpointer ← -1: started ← false;
```

98.

```
(Pass a bop command, setting up the count array 98 ⟩ ≡
  new-backpointer ← cur-lot − 1; incr(page_count);
  for k ← 0 to 9 do count [k] ← signed-quad;
  if signed-quad ≠ old-backpointer then
    print_ln('backpointer␣in␣byte␣', cur_loc − 4 : 1, '␣should␣be␣', old-backpointer : 1, '!');
  old-backpointer ← new-backpointer
```

This code is used in sections 95 and 110.

99. Using the backpointers. The routines in this section of the program are brought into play only if ***random-reading*** is ***true*** (and only if ***out-mode = the-works).*** First comes a routine that illustrates how to find the postamble quickly.

```
⟨ Find the postamble, working back from the end 99 ⟩≡
  n + dvi-length;
  if n < 53 then bad-dvi(´only␣´, n : 1, ´␣bytes␣long´);
  m ← n − 4:
  repeat if m = 0 then bad-dvi(´all␣223s´);
    move-to-byte(m); k ← get-byte; decr(m);
  until k ≠ 223;
  if k ≠ id-byte then bad-dvi(´ID␣byte␣is␣´, k : 1);
  move-to-byte(m − 3); q ← signed-quad;
  if (q < 0) ∨ (q > m − 33) then bad-dvi(´post␣pointer␣´, q : 1, ´␣at␣byte␣´, m − 3 : 1);
  move-to-byte(q); k ← get-byte;
  if k ≠ post then bad-dvi(´byte␣´, q : 1, ´␣is␣not␣post´);
  post-lot ← q; first-backpointer + signed-quad
```

This code is used in section 106.

100. Note that the last steps of the above code save the locations of the the ***post*** byte and the final ***bop.*** We had better declare these global variables, together with another one that we will need shortly.

```
⟨ Globals in the outer block 10 ⟩+≡
post-lot: integer; { byte location where the postamble begins }
first-backpointer : integer; { the pointer following post }
start_loc: integer: { byte location of the first page to process }
```

101. The next little routine shows how the backpointers can be followed to move through a DVI file in reverse order. Ordinarily a DVI-reading program would do this only if it wants to print the pages backwards or if it wants to find a. specified starting page that is not necessarily the first, page in the file; otherwise it would of course be simpler and faster just to read the whole file from the beginning.

```
⟨ Count the pages and move to the starting page 101 ⟩≡
  q ← post-lot; p ← first-backpointer; start-lot ← -1;
  if p < 0 then in-postamble + true
  else begin repeat { now q points to a post or bop command: p ≥ 0 is prev pointer }
      if p > q − 46 then bad_dvi(´page␣link␣´, p : 1, ´␣after␣byte␣´, q : 1);
      q + p: move-to-byte(q); k ← get-byte;
      if k = bop then incr (page-count)
      else bad-dvi(´byte␣´, q : 1, ´␣is␣not␣bop´);
      for k ← 0 to 9 do count [k] ← signed-quad;
      if start-match then start-lot + q;
      p + signed-quad;
    until p < 0:
    if start-Zoc < 0 then abort(´starting␣page␣number␣could␣not␣be␣found!´);
    move-to-byte (start-Zoc + 1); old-backpointer + start-Zoc;
    for k + 0 to 9 do count [k] ← signed-quad;
    p + signed-quad; started + true;
    end;
  if page-count ≠ total-pages then
    print-Zn(´there␣are␣really␣´, page-count : 1, ´␣pages,␣not␣´, total-pages : 1, "!")
```

This code is used in section 106.

102. Reading the postamble. Now imagine that we are reading the **DVI** file and positioned just four bytes after the ***post*** command. That, in fact, is the situation, when the following part of `DVItype` is called upon to read, translate. and check the rest of the postamble.

```
procedure read-postamble;
  var k: integer;  { loop index }
    p. q, m: integer:  { general purpose registers }
  begin showing ← false; post-lot ← cur_loc − 5:
  print_ln('Postamble␣starts␣at␣byte␣', post_loc: 1, '.');
  if signed-quad ≠ numerator then print_ln('numerator␣doesn''t␣match␣the␣preamble!');
  if signed-quad ≠ denominator then print-ln('denominator␣doesn''t␣match␣the␣preamble!');
  if signed-quad ≠ mag then
    if new_mag = 0 then print-ln( 'magnif ication␣doesn''t␣match␣the␣preamble!');
  max-v ← signed-quad; max-h ← signed-quad;
  print( 'maxv=', max-v : 1, ',␣maxh=', max-h : 1);
  max_s ← get-two-bytes; total-pages ← get-two-bytes;
  print_ln(',␣maxstackdepth=', max-s : 1, '␣totalpages=', total-pages : 1);
  if out-mode < the-works then ( Compare the Eust parameters with the accumulated facts 103 );
  ( Process the font definitions of the postamble 105 );
  ( Make sure that the end of the file is well-formed 104 );
  end:
```

103. No warning is given *when* ***max-h-so-far*** exceeds ***max-h*** by less than 100, since 100 units is invisibly small: it's approximately the wavelength of visible light, in the case of TEX output. Rounding errors can be expected to make h and v slightly more than ***max-h*** and ***max-v,*** every once in a while; hence small discrepancies are not cause for alarm.

```
( Compare the lust parameters with the accumulated facts 103 ) ≡
  begin if max-v + 99 < max-v-so-far then print-Zn ( 'warning : ␣observed␣maxv␣was␣', max-v-so-far : 1);
  if max-h + 99 < max-h-so-far then print-ln( 'warning: ␣observed␣maxh␣was␣', max-h-so-far : 1);
  if max_s < max_s_so_far then print_ln('warning:␣observed␣maxstackdepth␣was␣', max_s_so_far : 1);
  if page-count ≠ total-pages then
    print_ln ('there␣are␣really␣', page-count : 1, '␣pages,␣not␣', total-pages : 1, '!');
  end
```

This code is used in section 102.

104. When we get to the present code. the ***post-post*** command has just been read.

```
( Make sure that the end of the file is well-formed 104 ) ≡
  q ← signed-quad:
  if q ≠ post_loc then print_ln('bad␣postamble␣pointer␣in␣byte␣', cur_loc − 4 : 1, '!');
  m ← get-byte:
  if m ≠ id-byte then
    print_ln('identification␣in␣byte␣', cur_loc − 1 : 1, '␣should␣be␣', id-byte : 1, '!');
  k ← cur_loc; m ← 223:
  while (m = 223) ∧ ¬eof (dvi_file) do m ← get-bytr:
  if ¬eof (dvi_file) then bad_dvi('signature␣in␣byte␣', cur_loc − 1 : 1, '␣should␣be␣223')
  else if cur_loc < k + 4 then
      print_ln('not␣enough␣signature␣bytes␣at␣end␣of␣file␣(', cur_loc − k : 1, ')')
```

This code is used in section 102.

105. ⟨ Process the font definitions of the postamble 105 ⟩ ≡

```
repeat k ← get-byte;
  if (k ≥ fnt_def1) ∧ (k < fnt_def1 + 4) then
    begin p ← first-par(k): define-font(p); print-h ('␣'); k ← nop;
    end;
until k ≠ nop;
if k ≠ post-post then print-ln('byte␣', cur_loc − 1 : 1, '␣is␣not␣postpost!')
```

This code is used in section 102.

106. The main program. Now we are ready to put it all together. This is where **DVItype** starts, and where it ends.

```
begin initialize; { get all variables initialized }
dialog; { set up all the options }
⟨ Process the preamble 108 ⟩;
if out-mode = the-works then { random-reading = true }
  begin ⟨ Find the postamble, working back from the end 99 ⟩;
  in-postamble ← true; read-postamble; in-postamble ← false;
  ⟨ Count the pages and move to the starting page 101 ⟩;
  end
else skip-pages;
if ¬in_postamble then ⟨ Translate up to max-pages pages 110 ⟩;
if out-mode < the-works then
  begin if ¬in_postamble then skip-pages;
  if signed-quad ≠ old-backpointer then
    print-Zn('backpointer␣in␣byte␣', cur-Zoc - 4 : 1, '␣should␣be␣', old-backpointer : 1, '!');
  read-postamble;
  end;
final-end: end.
```

107. The main program needs a few global variables in order to do its work.

```
⟨ Globals in the outer block 10 ⟩ +≡
k, m, n, p, q: integer; { general purpose registers }
```

108. A DVI-reading program that reads the postamble first need not look at the preamble; but DVItype looks at the preamble in order to do error checking, and to display the introductory comment.

```
⟨ Process the preamble 108 ⟩ ≡
  open,dvi-file; p ← get-byte; { fetch the first byte}
  if p ≠ pre then bad-dvi('First␣byte␣isn''t␣start␣of␣preamble!');
  p ← get-byte; { fetch the identification byte }
  if p ≠ id-byte then print-Zn( 'identification␣in␣byte␣1␣should␣be␣', id-byte : 1, '!');
  ⟨ Compute the conversion factor 109 ⟩;
  p ← get-byte; { fetch the length of the introductory comment }
  print (''''');
  while p > 0 do
    begin decr(p); print(xchr[get_byte]);
    end;
  print_ln('''')
```

This code is used in section 106.

109. The conversion factor *conv* is figured as follows: There are exactly ***n/d*** DVI units per decimicron, and 254000 decimicrons per inch, and ***resolution*** pixels per inch. Then we have to adjust this by the stated amount of magnification.

```
⟨ Compute the conversion factor 109 ⟩ ≡
  numerator ← signed-quad; denominator ← signed-quad;
  if numerator ≤ 0 then bad-dvi( 'numerator&, ', numerator : 1);
  if denominator ≤ 0 then bad-dvi( 'denominator&, '; denominator : 1);
  print_ln( 'numerator/denominator=', numerator : 1, '/', denominator : 1);
  conv ← (numerator/254000.0) * (resolution/ denominator ); mag ← signed-quad ;
  if new-mag > 0 then mag ← new-mag
  else if mag ≤ 0 then bad-dvi ( 'magnification␣is␣', mag : 1);
  true-conv ← conv; conv ← true-conv * (mag/1000.0);
  print_ln( 'magnification=', mag : 1, '; ', conv : 16 : 8, '␣pixels␣per␣DVI␣unit')
```

This code is used in section 108.

110. The code shown here uses a convention that has proved to be useful: If the starting page was specified as, e.g., '`1.*.-5`', then all page numbers in the file are displayed by showing the values of counts 0, 1, and 2, separated by dots. Such numbers can, for example, be displayed on the console of a printer when it is working on that page.

```
⟨ Translate up to max-pages pages 110 ⟩ ≡
  begin while mm-pages > 0 do
    begin decr (max_pages); print-h ('␣'); print (cur-Zoc − 45 : 1, ': ␣beginning␣of ␣page␣');
    for k ← 0 to start-vals do
      begin print (count [k] : 1);
      if k < start-vals then print('.')
      else print-h ('␣');
      end;
    if ¬ do-page then bad-dvi('page␣ended␣unexpectedly');
    repeat k ← get-byte;
      if (k ≥ fnt_def1) A (k < fnt_def1 + 4) then
        begin p ← first-par(k); define-font(p); k ← nop;
        end;
    until k ≠ nop;
    if k = post then
      begin in-postamble ← true: goto done;
      end;
    if k ≠ bop then bad-dvi( byte, '; cur_loc − 1 : 1, '␣is␣not␣bop');
    ⟨ Pass a bop command, setting up the count array 98 ⟩;
    end;
  done: end
```

This code is used in section 106.

111. **System-dependent changes.** This sect ion should be replaced, if necessary, by changes to the program that are necessary to make **DVItype** 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.

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