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 *vf_file* and *tfm_file*; the output is written on *vpl_file*. Error messages and other remarks are written on the *output* file, which the user may choose to assign to the terminal if the system permits it.

The term *print* is used instead of *write* when this program writes on the *output* file, so that all such output can be easily deflected.

```
define print(#) ≡ write(stderr,#)
define print_ln(#) ≡ write_ln(stderr,#)
define print_real(#) ≡ fprint_real(stderr,#)
program VFtoVP(vf_file,tfm_file,vpl_file,output);
  label ⟨ Labels in the outer block 3 ⟩
  const ⟨ Constants in the outer block 4* ⟩
  type ⟨ Types in the outer block 5 ⟩
  var ⟨ Globals in the outer block 7 ⟩
    ⟨ Define parse_arguments 136* ⟩
  procedure initialize; { this procedure gets things started properly }
    var k: integer; { all-purpose index for initialization }
    begin kpse_set_progname(argv[0]); kpse_init_prog('VFTOVP',0,nil,nil); parse_arguments;
    ⟨ Set initial values 11* ⟩
    end;
```

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

```
define class ≡ class_var
⟨ Constants in the outer block 4* ⟩ ≡
  tfm_size = 150000; { maximum length of tfm data, in bytes }
  vf_size = 100000; { maximum length of vf data, in bytes }
  max_fonts = 300; { maximum number of local fonts in the vf file }
  lig_size = 32510; { maximum length of lig_kern program, in words }
  hash_size = 32579;
      { preferably a prime number, a bit larger than the number of character pairs in lig/kern steps }
  max_stack = 100; { maximum depth of DVI stack in character packets }
```

See also section 144*.

This code is used in section 2*.

11:* We don't have to do anything special to read a packed file of bytes, but we do want to use environment variables to find the input files.

⟨ Set initial values 11* ⟩ ≡
{ See comments at *kpse_find_vf* in `kpathsea/tex-file.h` for why we don't use it. }
vf_file ← *kpse_open_file*(*vf_name*, *kpse_vf_format*); *tfm_file* ← *kpse_open_file*(*tfm_name*, *kpse_tfm_format*);
if *verbose* **then**
begin *print*(*banner*); *print_ln*(*version_string*);
end;

See also sections 21*, 51*, 56, 69, and 87.

This code is used in section 2*.

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

⟨ Set initial values 11* ⟩ +≡
if *optind* + 3 > *argc* **then**
begin *vpl_file* ← *stdout*;
end
else begin *vpl_name* ← *extend_filename*(*cmdline*(*optind* + 2), `'vpl'`); *rewrite*(*vpl_file*, *vpl_name*);
end;

22* Unpacking the TFM file. The first thing `VFtoVP` does is read the entire *tfm_file* into an array of bytes, $tfm[0 \,..\, (4 * lf - 1)]$.

define *index* ≡ *index_type*

⟨ Types in the outer block 5 ⟩ +≡
index = 0 .. *tfm_size*; { address of a byte in *tfm* }

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

define *abort*(#) ≡
begin *print_ln*(#);
print_ln(´Sorry,␣but␣I␣can´´t␣go␣on;␣are␣you␣sure␣this␣is␣a␣TFM?´); *uexit*(1);
end

⟨ Read the whole `TFM` file 24* ⟩ ≡
read(*tfm_file*, *tfm*[0]);
if *tfm*[0] > 127 **then** *abort*(´The␣first␣byte␣of␣the␣input␣file␣exceeds␣127!´);
if *eof*(*tfm_file*) **then** *abort*(´The␣input␣file␣is␣only␣one␣byte␣long!´);
read(*tfm_file*, *tfm*[1]); *lf* ← *tfm*[0] ∗ ´400 + *tfm*[1];
if *lf* = 0 **then** *abort*(´The␣file␣claims␣to␣have␣length␣zero,␣but␣that´´s␣impossible!´);
if 4 ∗ *lf* − 1 > *tfm_size* **then** *abort*(´The␣file␣is␣bigger␣than␣I␣can␣handle!´);
for *tfm_ptr* ← 2 **to** 4 ∗ *lf* − 1 **do**
begin if *eof*(*tfm_file*) **then** *abort*(´The␣file␣has␣fewer␣bytes␣than␣it␣claims!´);
read(*tfm_file*, *tfm*[*tfm_ptr*]);
end;
if ¬*eof*(*tfm_file*) **then**
begin *print_ln*(´There´´s␣some␣extra␣junk␣at␣the␣end␣of␣the␣TFM␣file,´);
print_ln(´but␣I´´ll␣proceed␣as␣if␣it␣weren´´t␣there.´);
end

This code is used in section 132*.

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

define *eval_two_bytes*(#) ≡
begin if *tfm*[*tfm_ptr*] > 127 **then** *abort*(´One␣of␣the␣subfile␣sizes␣is␣negative!´);
← *tfm*[*tfm_ptr*] ∗ ´400 + *tfm*[*tfm_ptr* + 1]; *tfm_ptr* ← *tfm_ptr* + 2;
end

⟨ Set subfile sizes *lh*, *bc*, ..., *np* 25* ⟩ ≡
begin *tfm_ptr* ← 2;
eval_two_bytes(*lh*); *eval_two_bytes*(*bc*); *eval_two_bytes*(*ec*); *eval_two_bytes*(*nw*); *eval_two_bytes*(*nh*);
eval_two_bytes(*nd*); *eval_two_bytes*(*ni*); *eval_two_bytes*(*nl*); *eval_two_bytes*(*nk*); *eval_two_bytes*(*ne*);
eval_two_bytes(*np*);
if *lh* < 2 **then** *abort*(´The␣header␣length␣is␣only␣´, *lh* : 1, ´!´);
if *nl* > *lig_size* **then** *abort*(´The␣lig/kern␣program␣is␣longer␣than␣I␣can␣handle!´);
if (*bc* > *ec* + 1) ∨ (*ec* > 255) **then**
abort(´The␣character␣code␣range␣´, *bc* : 1, ´..´, *ec* : 1, ´is␣illegal!´);
if (*nw* = 0) ∨ (*nh* = 0) ∨ (*nd* = 0) ∨ (*ni* = 0) **then**
abort(´Incomplete␣subfiles␣for␣character␣dimensions!´);
if *ne* > 256 **then** *abort*(´There␣are␣´, *ne* : 1, ´␣extensible␣recipes!´);
if *lf* ≠ 6 + *lh* + (*ec* − *bc* + 1) + *nw* + *nh* + *nd* + *ni* + *nl* + *nk* + *ne* + *np* **then**
abort(´Subfile␣sizes␣don´´t␣add␣up␣to␣the␣stated␣total!´);
end

This code is used in section 132*.

31.* Again we cautiously verify that we've been given decent data.

```
define read_vf(#) ≡ read(vf_file, #)
define vf_abort(#) ≡
        begin print_ln(#); print_ln('Sorry, but I can''t go on; are you sure this is a VF?');
        uexit(1);
        end
```

⟨ Read the whole VF file 31* ⟩ ≡

```
read_vf(temp_byte);
if temp_byte ≠ pre then vf_abort('The first byte isn''t `pre''!');
⟨ Read the preamble command 32* ⟩;
⟨ Read and store the font definitions and character packets 33 ⟩;
⟨ Read and verify the postamble 34 ⟩
```

This code is used in section 132*.

32.*

```
define vf_store(#) ≡
        if vf_ptr + # ≥ vf_size then vf_abort('The file is bigger than I can handle!');
        for k ← vf_ptr to vf_ptr + # − 1 do
          begin if eof(vf_file) then vf_abort('The file ended prematurely!');
          read_vf(vf[k]);
          end;
        vf_count ← vf_count + #; vf_ptr ← vf_ptr + #
```

⟨ Read the preamble command 32* ⟩ ≡

```
if eof(vf_file) then vf_abort('The input file is only one byte long!');
read_vf(temp_byte);
if temp_byte ≠ id_byte then vf_abort('Wrong VF version number in second byte!');
if eof(vf_file) then vf_abort('The input file is only two bytes long!');
read_vf(temp_byte);   { read the length of introductory comment }
vf_count ← 11; vf_ptr ← 0; vf_store(temp_byte);
if verbose then
  begin for k ← 0 to vf_ptr − 1 do print(xchr[vf[k]]);
  print_ln(' ');
  end;
count ← 0;
for k ← 0 to 7 do
  begin if eof(vf_file) then vf_abort('The file ended prematurely!');
  read_vf(temp_byte);
  if temp_byte = tfm[check_sum + k] then incr(count);
  end;
real_dsize ← (((tfm[design_size] * 256 + tfm[design_size + 1]) * 256 + tfm[design_size + 2]) * 256 +
    tfm[design_size + 3])/'4000000;
if count ≠ 8 then
  begin print_ln('Check sum and/or design size mismatch.');
  print_ln('Data from TFM file will be assumed correct.');
  end
```

This code is used in section 31*.

35* ⟨ Read and store a font definition 35* ⟩ ≡
```
begin if packet_found ∨ (temp_byte ≥ pre) then
  vf_abort('Illegal byte', temp_byte : 1, ' at beginning of character packet!');
font_number[font_ptr] ← vf_read(temp_byte − fnt_def1 + 1);
if font_ptr = max_fonts then vf_abort('I can't handle that many fonts!');
vf_store(14);  { c[4] s[4] d[4] a[1] l[1] }
if vf[vf_ptr − 10] > 0 then  { s is negative or exceeds 2^24 − 1 }
  vf_abort('Mapped font size is too big!');
a ← vf[vf_ptr − 2]; l ← vf[vf_ptr − 1]; vf_store(a + l);  { n[a + l] }
if verbose then
  begin ⟨ Print the name of the local font 36* ⟩;
  end;
⟨ Read the local font's TFM file and record the characters it contains 39* ⟩;
incr(font_ptr); font_start[font_ptr] ← vf_ptr;
end
```
This code is used in section 33.

36* The font area may need to be separated from the font name on some systems. Here we simply reproduce the font area and font name (with no space or punctuation between them).

⟨ Print the name of the local font 36* ⟩ ≡
```
print('MAPFONT ', font_ptr : 1, ' : ');
for k ← font_start[font_ptr] + 14 to vf_ptr − 1 do print(xchr[vf[k]]);
k ← font_start[font_ptr] + 5; print(' at ');
print_real((((vf[k] * 256 + vf[k + 1]) * 256 + vf[k + 2])/'4000000) * real_dsize, 2, 2); print_ln('pt')
```
This code is used in section 35*.

37* Now we must read in another TFM file. But this time we needn't be so careful, because we merely want to discover which characters are present. The next few sections of the program are copied pretty much verbatim from DVItype, so that system-dependent modifications can be copied from existing software.

It turns out to be convenient to read four bytes at a time, when we are inputting from the local 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 7 ⟩ +≡
```
a: integer;  { length of the area/directory spec }
l: integer;  { length of the font name proper }
cur_name: ↑char;  { external tfm name }
b0, b1, b2, b3: byte;  { four bytes input at once }
font_lh: 0 .. '77777;  { header length of current local font }
font_bc, font_ec: 0 .. '77777;  { character range of current local font }
```

39* We use the *vf* array to store a list of all valid characters in the local font, beginning at location *font_chars*[*f*].

```
⟨ Read the local font's TFM file and record the characters it contains 39* ⟩ ≡
  font_chars[font_ptr] ← vf_ptr; ⟨ Move font name into the cur_name string 44* ⟩;
  tfm_name ← kpse_find_tfm(cur_name);
  if ¬tfm_name then
    print_ln('---not␣loaded,␣TFM␣file␣', stringcast(cur_name), '␣can''t␣be␣opened!')
  else begin resetbin(tfm_file, tfm_name); font_bc ← 0; font_ec ← 256;
        { will cause error if not modified soon }
    read_tfm_word;
    if b2 < 128 then
      begin font_lh ← b2 * 256 + b3; read_tfm_word;
      if (b0 < 128) ∧ (b2 < 128) then
        begin font_bc ← b0 * 256 + b1; font_ec ← b2 * 256 + b3;
        end;
      end;
    if font_bc ≤ font_ec then
      if font_ec > 255 then print_ln('---not␣loaded,␣bad␣TFM␣file␣', stringcast(tfm_name), '!')
      else begin for k ← 0 to 3 + font_lh do
          begin read_tfm_word;
          if k = 4 then ⟨ Check the check sum 40* ⟩;
          if k = 5 then ⟨ Check the design size 41 ⟩;
          end;
        for k ← font_bc to font_ec do
          begin read_tfm_word;
          if b0 > 0 then { character k exists in the font }
            begin vf[vf_ptr] ← k; incr(vf_ptr);
            if vf_ptr = vf_size then vf_abort('I''m␣out␣of␣VF␣memory!');
            end;
          end;
        end;
    if eof(tfm_file) then
      print_ln('---trouble␣is␣brewing,␣TFM␣file␣', stringcast(tfm_name), '␣ended␣too␣soon!');
    free(tfm_name);
    end;
  free(cur_name); incr(vf_ptr) { leave space for character search later }
```

This code is used in section 35*.

40* ⟨ Check the check sum 40* ⟩ ≡

```
  if b0 + b1 + b2 + b3 > 0 then
    if (b0 ≠ vf[font_start[font_ptr]]) ∨ (b1 ≠ vf[font_start[font_ptr] + 1]) ∨
        (b2 ≠ vf[font_start[font_ptr] + 2]) ∨ (b3 ≠ vf[font_start[font_ptr] + 3]) then
      begin if verbose then print_ln('Check␣sum␣in␣VF␣file␣being␣replaced␣by␣TFM␣check␣sum');
      vf[font_start[font_ptr]] ← b0; vf[font_start[font_ptr] + 1] ← b1; vf[font_start[font_ptr] + 2] ← b2;
      vf[font_start[font_ptr] + 3] ← b3;
      end
```

This code is used in section 39*.

43* (No initialization to be done. Keep this module to preserve numbering.)

44.* 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 44* ⟩ ≡

See also section 45*.

This code is used in section 39*.

45.* The string *cur_name* is supposed to be set to the external name of the `TFM` file for the current font. We do not impose an arbitrary limit on the filename length.

```
define name_start ≡ (font_start[font_ptr] + 14)
define name_end ≡ vf_ptr
```

⟨ Move font name into the *cur_name* string 44* ⟩ +≡

```
r ← name_end − name_start; cur_name ← xmalloc_array(char, r);
    { strncpy might be faster, but it's probably a good idea to keep the xchr translation. }
for k ← name_start to name_end do
  begin cur_name[k − name_start] ← xchr[vf[k]];
  end;
cur_name[r] ← 0;  { Append null byte since this is C. }
```

50.* In order to stick to standard Pascal, we use an *xchr* array to do appropriate conversion of ASCII codes. Three other little strings are used to produce *face* codes like `MIE`.

⟨ Globals in the outer block 7 ⟩ +≡
ASCII_04, *ASCII_10*, *ASCII_14*: *const_c_string*; { strings for output in the user's external character set }
xchr: **packed array** [0 .. 255] **of** *char*;
MBL_string, *RI_string*, *RCE_string*: *const_c_string*; { handy string constants for *face* codes }

51.* ⟨ Set initial values 11* ⟩ +≡
ASCII_04 ← ´␣␣!"#$%&´´()*+,-./0123456789:;<=>?´;
ASCII_10 ← ´␣@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_´;
ASCII_14 ← ´␣`abcdefghijklmnopqrstuvwxyz{|}~?´;
for $k \leftarrow 0$ **to** 255 **do** *xchr*[*k*] ← ´?´;
for $k \leftarrow 0$ **to** ´37 **do**
begin *xchr*[*k* + ´40] ← *ASCII_04*[*k* + 1]; *xchr*[*k* + ´100] ← *ASCII_10*[*k* + 1];
xchr[*k* + ´140] ← *ASCII_14*[*k* + 1];
end;
MBL_string ← ´␣MBL´; *RI_string* ← ´␣RI␣´; *RCE_string* ← ´␣RCE´;

61.* The property value may be a character, which is output in octal unless it is a letter or a digit.

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

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

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

63.* And finally, the value might be a *fix_word*, which is output in decimal notation with just enough decimal places for `VPtoVF` 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) * intcast('400) + tfm[k + 2]) * '400 + tfm[k + 3];
  if a > '3777 then ⟨ Reduce negative to positive 66 ⟩;
  ⟨ Output the integer part, a, in decimal notation 64 ⟩;
  ⟨ Output the fraction part, f/2^20, in decimal notation 65 ⟩;
  end;
```

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

```
⟨ Do the characters 101* ⟩ ≡
  sort_ptr ← 0;  { this will suppress 'STOP' lines in ligature comments }
  for c ← bc to ec do
    if width_index(c) > 0 then
      begin if chars_on_line = 8 then
        begin print_ln('␣'); chars_on_line ← 1;
        end
      else begin if chars_on_line > 0 then print('␣');
        if verbose then incr(chars_on_line);  { keep chars_on_line = 0 }
        end;
      if verbose then print_octal(c);  { progress report }
      left; out('CHARACTER'); out_char(c); out_ln; ⟨ Output the character's width 102 ⟩;
      if height_index(c) > 0 then ⟨ Output the character's height 103 ⟩;
      if depth_index(c) > 0 then ⟨ Output the character's depth 104 ⟩;
      if italic_index(c) > 0 then ⟨ Output the italic correction 105 ⟩;
      case tag(c) of
      no_tag: do_nothing;
      lig_tag: ⟨ Output the applicable part of the ligature/kern program as a comment 106 ⟩;
      list_tag: ⟨ Output the character link unless there is a problem 107 ⟩;
      ext_tag: ⟨ Output an extensible character recipe 108 ⟩;
      end;
      if ¬do_map(c) then goto final_end;
      right;
      end
```

This code is used in section 134*.

113* ⟨ Check for ligature cycles 113* ⟩ ≡

```
hash_ptr ← 0; y_lig_cycle ← 256;
for hh ← 0 to hash_size do hash[hh] ← 0;  { clear the hash table }
for c ← bc to ec do
  if tag(c) = lig_tag then
    begin i ← remainder(c);
    if tfm[lig_step(i)] > stop_flag then i ← 256 * tfm[lig_step(i) + 2] + tfm[lig_step(i) + 3];
    ⟨ Enter data for character c starting at location i in the hash table 114 ⟩;
    end;
if bchar_label < nl then
  begin c ← 256; i ← bchar_label;
  ⟨ Enter data for character c starting at location i in the hash table 114 ⟩;
  end;
if hash_ptr = hash_size then
  begin print_ln('Sorry, I haven't room for so many ligature/kern pairs!'); uexit(1);
  end;
for hh ← 1 to hash_ptr do
  begin r ← hash_list[hh];
  if class[r] > simple then  { make sure f is defined }
    r ← lig_f(r, (hash[r] − 1) div 256, (hash[r] − 1) mod 256);
  end;
if y_lig_cycle < 256 then
  begin print('Infinite ligature loop starting with ');
  if x_lig_cycle = 256 then print('boundary') else print_octal(x_lig_cycle);
  print(' and '); print_octal(y_lig_cycle); print_ln('!');
  out('(INFINITE LIGATURE LOOP MUST BE BROKEN!)'); uexit(1);
  end
```

This code is used in section 89.

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

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

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

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

125* ⟨ Do the packet for character c 125* ⟩ ≡

```
if packet_start[c] = vf_size then bad_vf(´Missing␣packet␣for␣character␣´, c : 1)
else begin left; out(´MAP´); out_ln; top ← 0; wstack[0] ← 0; xstack[0] ← 0; ystack[0] ← 0;
  zstack[0] ← 0; vf_ptr ← packet_start[c]; vf_limit ← packet_end[c] + 1; f ← 0;
  while vf_ptr < vf_limit do
    begin o ← vf[vf_ptr]; incr(vf_ptr);
    if (o ≤ set1 + 3) ∨ ((o ≥ put1) ∧ (o ≤ put1 + 3)) then
      ⟨ Special cases of DVI instructions to typeset characters 130* ⟩
    else case o of
        ⟨ Cases of DVI instructions that can appear in character packets 127* ⟩
      improper_DVI_for_VF: bad_vf(´Illegal␣DVI␣code␣´, o : 1, ´␣will␣be␣ignored´);
      end; { there are no other cases }
    end;
  if top > 0 then
    begin bad_vf(´More␣pushes␣than␣pops!´);
    repeat out(´(POP)´); decr(top); until top = 0;
    end;
  right;
  end
```

This code is used in section 134*.

126* A procedure called *get_bytes* helps fetch the parameters of DVI commands.

```
  define signed ≡ is_signed { signed is a reserved word in ANSI C }
function get_bytes(k : integer; signed : boolean): integer;
  var a: integer; { accumulator }
  begin if vf_ptr + k > vf_limit then
    begin bad_vf(´Packet␣ended␣prematurely´); k ← vf_limit − vf_ptr;
    end;
  a ← vf[vf_ptr];
  if (k = 4) ∨ signed then
    if a ≥ 128 then a ← a − 256;
  incr(vf_ptr);
  while k > 1 do
    begin a ← a ∗ 256 + vf[vf_ptr]; incr(vf_ptr); decr(k);
    end;
  get_bytes ← a;
  end;
```

127* Let's look at the simplest cases first, in order to get some experience.

```
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)
```

⟨ Cases of DVI instructions that can appear in character packets 127* ⟩ ≡

```
nop: do_nothing;
push: begin if top = max_stack then
    begin print_ln('Stack␣overflow!'); uexit(1);
    end;
  incr(top); wstack[top] ← wstack[top − 1]; xstack[top] ← xstack[top − 1]; ystack[top] ← ystack[top − 1];
  zstack[top] ← zstack[top − 1]; out('(PUSH)'); out_ln;
  end;
pop: if top = 0 then bad_vf('More␣pops␣than␣pushes!')
  else begin decr(top); out('(POP)'); out_ln;
    end;
set_rule, put_rule: begin if o = put_rule then out('(PUSH)');
  left; out('SETRULE'); out_as_fix(get_bytes(4, true)); out_as_fix(get_bytes(4, true));
  if o = put_rule then out(')(POP');
  right;
  end;
```

See also sections 128, 129, and 131.

This code is used in section 125*.

130* Before we typeset a character we make sure that it exists.

⟨ Special cases of DVI instructions to typeset characters 130* ⟩ ≡

```
begin if o ≥ set1 then
    if o ≥ put1 then k ← get_bytes(o − put1 + 1, false)
    else k ← get_bytes(o − set1 + 1, false)
  else k ← o;
  c ← k;
  if (k < 0) ∨ (k > 255) then bad_vf('Character␣', k : 1, '␣is␣out␣of␣range␣and␣will␣be␣ignored')
  else if f = font_ptr then bad_vf('Character␣', c : 1, '␣in␣undeclared␣font␣will␣be␣ignored')
    else begin vf[font_start[f + 1] − 1] ← c;  { store c in the "hole" we left }
      k ← font_chars[f]; while vf[k] ≠ c do incr(k);
      if k = font_start[f + 1] − 1 then
        bad_vf('Character␣', c : 1, '␣in␣font␣', f : 1, '␣will␣be␣ignored')
      else begin if o ≥ put1 then out('(PUSH)');
        left; out('SETCHAR'); out_char(c);
        if o ≥ put1 then out(')(POP');
        right;
        end;
      end;
  end
```

This code is used in section 125*.

132* 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 come the *vf_input* and *organize* procedures, which read the input data and get ready for subsequent events. If something goes wrong, the routines return *false*.

```
function vf_input: boolean;
  var vf_ptr: 0 .. vf_size;  { an index into vf }
    k: integer;  { all-purpose index }
    c: integer;  { character code }
  begin ⟨ Read the whole VF file 31* ⟩;
  vf_input ← true;
  end;
function organize: boolean;
  var tfm_ptr: index;  { an index into tfm }
  begin ⟨ Read the whole TFM file 24* ⟩;
  ⟨ Set subfile sizes lh, bc, ..., np 25* ⟩;
  ⟨ Compute the base addresses 27 ⟩;
  organize ← vf_input;
  end;
```

134* And then there's a routine for individual characters.

```
function do_map(c : byte): boolean;
  var k: integer; f: 0 .. vf_size;  { current font number }
  begin ⟨ Do the packet for character c 125* ⟩;
  do_map ← true;
  end;
function do_characters: boolean;
  label final_end, exit;
  var c: byte;  { character being done }
    k: index;  { a random index }
    ai: 0 .. lig_size;  { index into activity }
  begin ⟨ Do the characters 101* ⟩;
  do_characters ← true; return;
final_end: do_characters ← false;
exit: end;
```

135* Here is where VFtoVP begins and ends.

```
  begin initialize;
  if ¬organize then goto final_end;
  do_simple_things;
  ⟨ Do the ligatures and kerns 89 ⟩;
  ⟨ Check the extensible recipes 110 ⟩;
  if ¬do_characters then goto final_end;
  if verbose then print_ln('.');
  if level ≠ 0 then print_ln('This␣program␣isn''t␣working!');
  if ¬perfect then
    begin out('(COMMENT␣THE␣TFM␣AND/OR␣VF␣FILE␣WAS␣BAD,␣');
    out('SO␣THE␣DATA␣HAS␣BEEN␣CHANGED!)'); write_ln(vpl_file);
    end;
final_end: end.
```

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

```
define argument_is(#) ≡ (strcmp(long_options[option_index].name, #) = 0)
⟨ Define parse_arguments 136* ⟩ ≡
procedure parse_arguments;
  const n_options = 4;  { Pascal won't count array lengths for us. }
  var long_options: array [0 .. n_options] of getopt_struct;
    getopt_return_val: integer; option_index: c_int_type; current_option: 0 .. n_options;
  begin ⟨ Initialize the option variables 141* ⟩;
  ⟨ Define the option table 137* ⟩;
  repeat getopt_return_val ← getopt_long_only(argc, argv, ´´, long_options, address_of(option_index));
    if getopt_return_val = −1 then
      begin  { End of arguments; we exit the loop below. }
      ;
      end
    else if getopt_return_val = "?" then
        begin usage(´vftovp´);
        end
      else if argument_is(´help´) then
          begin usage_help(VFTOVP_HELP, nil);
          end
        else if argument_is(´version´) then
            begin print_version_and_exit(banner, nil, ´D.E. Knuth´, nil);
            end
          else if argument_is(´charcode-format´) then
              begin if strcmp(optarg, ´ascii´) = 0 then charcode_format ← charcode_ascii
              else if strcmp(optarg, ´octal´) = 0 then charcode_format ← charcode_octal
                else print_ln(´Bad character code format´, optarg, ´.´);
              end;  { Else it was a flag; getopt has already done the assignment. }
  until getopt_return_val = −1;  { Now optind is the index of first non-option on the command line. We
        must have one two three remaining arguments. }
  if (optind + 1 ≠ argc) ∧ (optind + 2 ≠ argc) ∧ (optind + 3 ≠ argc) then
    begin print_ln(´vftovp: Need one to three file arguments.´); usage(´vftovp´);
    end;
  vf_name ← cmdline(optind);
  if optind + 2 ≤ argc then
    begin tfm_name ← cmdline(optind + 1);  { The user specified the TFM name. }
    end
  else begin  { User did not specify TFM name; default it from the VF name. }
    tfm_name ← basename_change_suffix(vf_name, ´.vf´, ´.tfm´);
    end;
  end;
```

This code is used in section 2*.

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

```
⟨ Define the option table 137* ⟩ ≡
  current_option ← 0; long_options[current_option].name ← ´help´;
  long_options[current_option].has_arg ← 0; long_options[current_option].flag ← 0;
  long_options[current_option].val ← 0; incr(current_option);
```

See also sections 138*, 139*, 142*, and 147*.

This code is used in section 136*.

138.* Another of the standard options.

⟨ Define the option table 137* ⟩ +≡
long_options[*current_option*].*name* ← ´`version`´; *long_options*[*current_option*].*has_arg* ← 0;
long_options[*current_option*].*flag* ← 0; *long_options*[*current_option*].*val* ← 0; *incr*(*current_option*);

139.* Print progress information?

⟨ Define the option table 137* ⟩ +≡
long_options[*current_option*].*name* ← ´`verbose`´; *long_options*[*current_option*].*has_arg* ← 0;
long_options[*current_option*].*flag* ← *address_of*(*verbose*); *long_options*[*current_option*].*val* ← 1;
incr(*current_option*);

140.* The global variable *verbose* determines whether or not we print progress information.

⟨ Globals in the outer block 7 ⟩ +≡
verbose: *c_int_type*;

141.* It starts off *false*.

⟨ Initialize the option variables 141* ⟩ ≡
verbose ← *false*;
See also section 146*.
This code is used in section 136*.

142.* Here is an option to change how we output character codes.

⟨ Define the option table 137* ⟩ +≡
long_options[*current_option*].*name* ← ´`charcode-format`´; *long_options*[*current_option*].*has_arg* ← 1;
long_options[*current_option*].*flag* ← 0; *long_options*[*current_option*].*val* ← 0; *incr*(*current_option*);

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

⟨ Types in the outer block 5 ⟩ +≡
charcode_format_type = *charcode_ascii* .. *charcode_default*;

144.*

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

145.*

⟨ Globals in the outer block 7 ⟩ +≡
charcode_format: *charcode_format_type*;

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

⟨ Initialize the option variables 141* ⟩ +≡
charcode_format ← *charcode_default*;

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

⟨ Define the option table 137* ⟩ +≡
long_options[*current_option*].*name* ← 0; *long_options*[*current_option*].*has_arg* ← 0;
long_options[*current_option*].*flag* ← 0; *long_options*[*current_option*].*val* ← 0;

148.* Global filenames.

⟨ Globals in the outer block 7 ⟩ +≡
vf_name, *tfm_name*, *vpl_name*: *c_string*;

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

The following sections were changed by the change file: 2, 4, 11, 21, 22, 24, 25, 31, 32, 35, 36, 37, 39, 40, 43, 44, 45, 50, 51, 61, 62, 63, 101, 113, 117, 118, 125, 126, 127, 130, 132, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149.

⟨ Output the rest of the header 79 ⟩ Used in section 71.
⟨ Output the *seven_bit_safe_flag* 80 ⟩ Used in section 71.
⟨ Print the name of the local font 36* ⟩ Used in section 35*.
⟨ Read and store a character packet 47 ⟩ Used in section 33.
⟨ Read and store a font definition 35* ⟩ Used in section 33.
⟨ Read and store the font definitions and character packets 33 ⟩ Used in section 31*.
⟨ Read and verify the postamble 34 ⟩ Used in section 31*.
⟨ Read the local font's `TFM` file and record the characters it contains 39* ⟩ Used in section 35*.
⟨ Read the preamble command 32* ⟩ Used in section 31*.
⟨ Read the whole `TFM` file 24* ⟩ Used in section 132*.
⟨ Read the whole `VF` file 31* ⟩ Used in section 132*.
⟨ Reduce l by one, preserving the invariants 60 ⟩ Used in section 59.
⟨ Reduce negative to positive 66 ⟩ Used in section 63*.
⟨ Set initial values 11*, 21*, 51*, 56, 69, 87 ⟩ Used in section 2*.
⟨ Set subfile sizes lh, bc, ..., np 25* ⟩ Used in section 132*.
⟨ Set the true *font_type* 76 ⟩ Used in section 71.
⟨ Special cases of `DVI` instructions to typeset characters 130* ⟩ Used in section 125*.
⟨ Take care of commenting out unreachable steps 96 ⟩ Used in section 94.
⟨ Types in the outer block 5, 22*, 143* ⟩ Used in section 2*.

The VFtoVP processor

(Version 1.3, December 2002)

The preparation of this program was supported in part by the National Science Foundation and by the System Development Foundation. 'TEX' is a trademark of the American Mathematical Society.