The POOLtype processor

(Version 3, September 1989)

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 `TANGLE`d 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 256 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 eight-bit code that includes 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 . . 255; { eight-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 lowercase letters. Nowadays, of course, we need to deal with both capital and small letters 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* = 255 { ordinal number of the largest element of *text_char* }

⟨ Local variables for initialization 6 ⟩ ≡
i: *integer*;

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 the standard part 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] ← '~';
```

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 many systems prohibit in text files }
```

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*[0 .. ´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'. 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. To get the most "permissive" character set, change ´␣´ on the right of these assignment statements to *chr*(*i*).

```
⟨Set initial values of key variables 8⟩ +≡
  for $i \gets 0$ to ´37 do $xchr[i] \gets$ ´␣´;
  for $i \gets$ ´177 to ´377 do $xchr[i] \gets$ ´␣´;
```

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 $i \gets first\_text\_char$ to $last\_text\_char$ do $xord[chr(i)] \gets invalid\_code$;
  for $i \gets$ ´200 to ´377 do $xord[xchr[i]] \gets i$;
  for $i \gets 0$ to ´176 do $xord[xchr[i]] \gets i$;
```

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

```
⟨ Globals in the outer block 7 ⟩ +≡
k, l: 0 . . 255;  { 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_ln(#); goto 9999;
          end
  begin initialize;
  ⟨ Make the first 256 strings 16 ⟩;
  s ← 256;
  ⟨ Read the other strings from the POOL file, or give an error message and abort 19 ⟩;
  write_ln(´(´, count : 1, ´␣characters␣in␣all.)´);
9999: end.
```

16.

```
define lc_hex(#) ≡ l ← #;
    if l < 10 then l ← l + "0" else l ← l − 10 + "a"

⟨ Make the first 256 strings 16 ⟩ ≡
  for k ← 0 to 255 do
    begin write(k : 3, ´:␣"´); l ← k;
    if (⟨ Character k cannot be printed 17 ⟩) then
      begin write(xchr["^"], xchr["^"]);
      if k < ´100 then l ← k + ´100
      else if k < ´200 then l ← k − ´100
        else begin lc_hex(k div 16); write(xchr[l]); lc_hex(k mod 16); incr(count);
          end;
      count ← count + 2;
      end;
    if l = """" then write(xchr[l], xchr[l])
    else write(xchr[l]);
    incr(count); write_ln(´"´);
    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.

Unprintable characters of codes 128–255 are, similarly, rendered `^^80`–`^^ff`.

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. An appropriate formula for the extended character set recommended in *The TEXbook* would, for example, be '$k \in [0, ´10 \mathrel{..} ´12, ´14, ´15, ´33, ´177 \mathrel{..} ´377]$'. If character k cannot be printed, and $k < ´200$, then character $k + ´100$ or $k - ´100$ must be printable; moreover, ASCII codes $[´41 \mathrel{..} ´46, ´60 \mathrel{..} ´71, ´141 \mathrel{..} ´146, ´160 \mathrel{..} ´171]$ must be printable. Thus, at least 80 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_ln('"'); abort('!␣That␣POOL␣line␣was␣too␣short');
         end;
      read(pool_file, m); write(xchr[xord[m]]);
      if xord[m] = """" then write(xchr[""""]);
      end;
   write_ln('"'); incr(s);
   end
else xsum ← true;
read_ln(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.

⟨ Character k cannot be printed 17 ⟩ Used in section 16.
⟨ Globals in the outer block 7, 12, 13, 18 ⟩ Used in section 2.
⟨ Local variables for initialization 6 ⟩ Used in section 2.
⟨ Make the first 256 strings 16 ⟩ Used in section 15.
⟨ Read one string, but abort if there are problems 20 ⟩ Used in section 19.
⟨ Read the other strings from the `POOL` file, or give an error message and abort 19 ⟩ Used in section 15.
⟨ Set initial values of key variables 8, 10, 11, 14 ⟩ Used in section 2.
⟨ Types in the outer block 5 ⟩ Used in section 2.