The POOLtype processor

(Version 3, September 1989)

Editor's Note: The present variant of this C/WEB source file has been modified for use in the TEX Live system.

The following sections were changed by the change file: 1, 2, 6, 10, 15, 18, 19, 20, 21, 22, 23, 24, 25.

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.

```
define my_name ≡ 'pooltype'
```

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);
  type ⟨Types in the outer block 5⟩
  var ⟨Globals in the outer block 7⟩
    ⟨Define parse_arguments 21*⟩
  procedure initialize; { this procedure gets things started properly }
    var ⟨Local variables for initialization 6*⟩
    begin kpse_set_program_name(argv[0], my_name); parse_arguments;
    ⟨Set initial values of key variables 8⟩
    end;
```

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 ≡ ASCII_code   { 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*.

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 ← 0 to ´37 do xchr[i] ← chr(i);
for i ← ´177 to ´377 do xchr[i] ← chr(i);
```

15* This is the main program, where `POOLtype` starts and ends.

```
define abort(#) ≡
        begin write_ln(stderr, #); uexit(1);
        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.)'); uexit(0);
end.
```

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 }
pool_name: const_c_string;
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*⟩ ≡

```
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); read(pool_file, 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. Parse a Unix-style command line.

```
  define argument_is(#) ≡ (strcmp(long_options[option_index].name, #) = 0)
⟨ Define parse_arguments 21* ⟩ ≡
procedure parse_arguments;
  const n_options = 2;  { 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 ⟨ Define the option table 22* ⟩;
  repeat getopt_return_val ← getopt_long_only(argc, argv, ´´, long_options, address_of(option_index));
    if getopt_return_val = −1 then
      begin do_nothing;
      end
    else if getopt_return_val = ´?´ then
        begin usage(my_name);
        end
      else if argument_is(´help´) then
          begin usage_help(POOLTYPE_HELP, nil);
          end
        else if argument_is(´version´) then
            begin print_version_and_exit(´This␣is␣POOLtype,␣Version␣3.0´, nil, ´D.E.␣Knuth´, nil);
            end;  { Else it was just a flag; getopt has already done the assignment. }
  until getopt_return_val = −1;  { Now optind is the index of first non-option on the command line. }
  if (optind + 1 ≠ argc) then
    begin write_ln(stderr, my_name, ´:␣Need␣exactly␣one␣file␣argument.´); usage(my_name);
    end;
  pool_name ← extend_filename(cmdline(optind), ´pool´);
      { Try opening the file here, to avoid printing the first 256 strings if they give a bad filename. }
  resetbin(pool_file, pool_name);
  end;
```

This code is used in section 2*.

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

```
⟨ Define the option table 22* ⟩ ≡
  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 23* and 24*.

This code is used in section 21*.

23*. Another of the standard options.

```
⟨ Define the option table 22* ⟩ +≡
  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);
```

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

```
⟨ Define the option table 22* ⟩ +≡
  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;
```

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

The following sections were changed by the change file: 1, 2, 6, 10, 15, 18, 19, 20, 21, 22, 23, 24, 25.