Software

Exercises for TEX: *The Program*

Donald E. Knuth

During the spring of 1987 I taught a course for which Volume B of *Computers & Typesetting* was the textbook. Since that book was meant to serve primarily as a reference, not as a text, I needed to supplement it with homework exercises and exam problems.

The problems turned out to rather interesting, and they might be useful for self-study if anybody wants to learn *TEX: The Program* without taking a college course. Therefore I've collected them here and given what I think are the correct answers.

The final problem, which deals with the typesetting of languages that have large character sets, is especially noteworthy since it presents an extension of TEX that might prove to be useful in Asia.

Some of the problems suggested changes in the text. I've changed my original wording of the problem statements so that they will make sense when the next printing of the book comes out; people who have the first edition should check the published errata before looking too closely at the questions below. But some problems (e.g. 25, 26, and 32) assume the old 7-bit version of TEX.

Editor's note: The answers to the following exercises will appear in the November 1990 issue of *TUGboat*.

The Problems

Here, then, are the exercises in the order I gave them. Although they begin with a rather "gentle introduction," I recommend that the first ones not be skipped, even if they may appear too easy; there often is a slightly subtle point involved. Conversely, some of the problems are real stumpers, but they are intended to teach important lessons. A serious attempt should be made to solve each one before turning to the answer, if the maximum benefit is to be achieved.

1. (An exercise about reading a `WEB`.) In the Pascal program defined by the book, what immediately precedes '`PROCEDURE INITIALIZE`'? (Of course it's a semicolon, but you should also figure out a few things that occur immediately before that semicolon.)

2. Find an unnecessary macro in §15.

3. Suppose that you want to make TEX work in an environment where the input file can contain two-character sequences of the form '*esc x*', where *esc* is ASCII character number *'33* and where x is an ASCII character between `'@'` and `'_'` inclusive. The result should be essentially equivalent to what would have happened if the single (possibly nonprinting) ASCII character $chr(ord(x) - '100)$ had been input instead. If *esc* appears without being followed by an x in the desired range, you should treat it as if the *esc* were ASCII character number *'177*.

For example, the line '`A` *esc* `A` *esc* `a` *eoln*' should put four codes into the buffer: *'101*, *'001*, *'177*, *'141*.

What system-dependent changes will handle this interface requirement?

4. Suppose that the string at the beginning of the *print_roman_int* procedure were `"m2d5c2l2q5v5i"` instead of `"m2d5c2l5x2v5i"`. What would be printed from the input 69? From the input 9999?

5. Why does *error_count* have a lower bound of -1?

6. What is printed on the user's terminal after 'q' is typed in response to an error prompt? Why?

7. Give examples of how TEX might fail in the following circumstances:

a) If the test '$t \le 7230584$' were eliminated from §108.
b) If the test '$s \ge 1663497$' were eliminated from §108.
c) If the test '$r > p + 1$' were changed to '$r > p$' in §127.
d) If the test '$rlink(p) \ne p$' were eliminated from §127.
e) If the test '$lo\_mem\_max + 2 \le mem\_bot + max\_halfword$' were eliminated from §125.

8. The purpose of this problem is to figure out what data in *mem* could have generated the following output of *show_node_list*:

```
\hbox(10.0+0.0)x100.0, glue set 10.0fill         100
.\discretionary replacing 1                      200
..\kern 10.0                                     300
.|\large U                                     10000
.|\large ^^K (ligature ff)                       400, 10001, 10002
.\large !                                      10003
.\penalty 5000                                   500
.\glue 0.0 plus 1.0fill                          600
.\vbox(5.0x0.5)x10.0, shifted -5.0               700
..\hbox(5.0x0.0)x10.0                            800
...\small d                                    10004
...\small a                                    10005
..\rule(0.5+0.0)x*                               900
```

Assume that `\large` is font number 1 and that `\small` is font number 2. Also assume that the nodes used in the lower (variable-size) part of *mem* start in locations 100, 200, etc., as shown; the nodes used in the upper (one-word) part of *mem* should appear in locations 10000, 10001, etc. Make a diagram that illustrates the exact numeric contents of every relevant *mem* word.

9. What will *short_display* print, when given the horizontal list inside the larger `\hbox` in the previous problem, assuming that *font_in_short_display* is initially zero?

10. Suppose the following commands are executed immediately after TeX has initialized itself:

> *incr*(*prev_depth*);
> *decr*(*mode_line*);
> *incr*(*prev_graf*);
> *show_activities*.

What will be shown?

11. What will '*show_eqtb*(*int_base* + 17)' show, after TeX has initialized itself?

12. Suppose TeX has been given the following definitions:

```
\def\a{\advance\day by 1\relax}
\def\g{\global\a}
```

The effect of this inside TeX will be that an appearance of `\a` calls

$$eq\_word\_define(p, eqtb[p].int + 1),$$

and an appearance of `\g` calls

$$geq\_word\_define(p, eqtb[p].int + 1)$$

where $p = int\_base + day\_code$. Consider now the following commands:

```
\day=0 \g\a{\a\g\a{\g\a\g}\a{\a}\a}
```

Each '{' calls *new_save_level*(*simple_group*), and each '}' calls *unsave*.

Explain what gets pushed onto and popped off of the *save_stack*, and what gets stored in *eqtb*[*p*] and *xeq_level*[*p*], as the above commands are executed. What is the final value of `\day`? (See *The TeXbook*, exercise 15.9 and page 301.)

13. Use the notation at the bottom of page 122 in *TeX: The Program* to describe the contents of the token list corresponding to `\!` after the definition

```
\def\!!1#2![{!#]#!!2}
```

has been given, assuming that `[`, `]`, and `!` have the respective catcodes 1, 2, and 6, just as `{`, `}`, and `#` do. (See exercise 20.7 in *The TeXbook*.)

14. What is the absolute maximum number of characters that will printed by *show_eqtb*(*every_par_loc*), if the current value of `\everypar` does not contain any control sequences? (Hint: The answer exceeds 40. You may wish to verify this by running TeX, defining an appropriate worst-case example, and saying

```
\tracingrestores=1
\tracingonline=1
{\everypar{}}
```

since this will invoke *show_eqtb* when `\everypar` is restored.)

15. What does `INITEX` do with the following input line? (Look closely.)

```
\catcode‘‘=7 \‘‘ ‘(‘‘)‘‘‘!
```

16. Explain the error message you get if you say

```
\endlinechar=‘! \error
```

in plain TeX.

17. Fill in the missing macro definition so that the program

```
\catcode'?=\active
\def\answer{...}
\answer
```

will produce precisely the following error message when run with plain TeX:

```
! Undefined control sequence.
<recently read> How did this happen?

l.3 \answer

?
```

(This problem is much harder than the others above, but there are at least three ways to solve it!)

18. Consider what TeX will do when it processes the following text:

```
{\def\t{\gdef\a##}\catcode'd=12\t1d#2#3{#2}}
\hfuzz=100P\ifdim12pt=1P\expandafter\a
 \expandafter\else\romannumeral888\relax\fi
\showthe\hfuzz \showlists
```

(Assume that the category codes of plain TeX are being used.)

Determine when the subroutines *scan_keyword*, *scan_int*, and *scan_dimen* are called as this text is being read, and explain in general terms what results those subroutines produce.

19. What is the difference in interpretation, if any, between the following two TeX commands?

```
\thickmuskip=-\thickmuskip
\thickmuskip=-\the\thickmuskip
```

(Assume that plain TeX is being used.) Explain why there is or isn't a difference.

20. In what way would TeX's behavior change if the assignment at the end of §508 were changed to '$b \leftarrow (p = null)$'?

21. The initial implementation of TeX82 had a much simpler procedure in place of the one now in §601:

```
procedure dvi_pop;
  begin if dvi_ptr > 0 then
    if dvi_buf[dvi_ptr - 1] = push then decr(dvi_ptr)
    else dvi_out(pop)
  else dvi_out(pop);
  end;
```

(No parameter l was necessary.) Why did the author hang his head in shame one day and change it to the form it now has?

22. Assign subscripts d, y, and z to the sequence of integers

2 7 1 8 2 8 1 8 2 8 4 5 9 0 4 5

using the procedure sketched in §604. (This is easy.)

23. Find a short TeX program that will cause the *print_mode* subroutine to print '`no mode`'. (Do not assume that the category codes or macros of plain TeX have been preloaded.) Extra credit will be given to the person who has the shortest program, i.e., the fewest tokens, among all correct solutions submitted.

24. The textbook says in §78 that *error* might be called within *error* within a call of *error*, but the recursion cannot go any deeper than this.

Construct a scenario in which *error* is entered three times before it has been completed.

25. (The following question was the main problem on the midterm exam.) Suppose `WEB`'s conventions have been changed so that strings are not identified by their number but rather by their starting position in the *str_pool* array. The *str_start* array is therefore eliminated.

Strings of length 1 are still represented by their ASCII code values; all other strings have values ≥ 128, and they appear in the *pool_file* just as before, in increasing order of starting position. The special code '128' is assumed to terminate each string.

Thus, for example, if such a `WEB` program uses just the three strings `"ab"`, `""`, and `"cd"` (in this order), they will be represented in the corresponding Pascal code by the respective integers 128, 131, and 132. The program in this case is expected to initialize *str_pool* locations 128–134 to the successive code values 97, 98, 128, 128, 99, 100, 128.

Such a modification requires lots of changes to TeX. Your job in this problem is to indicate what those changes should be. However, you needn't specify a complete change file; just say how you would modify §38–§48, §59, §259, §407, §464, and §602 (if these sections need to change at all). The other places where *str_start* appears can be changed in similar ways, and you needn't deal with those.

Some of the specified sections will require new code; you should supply that code. Other sections may change only a little bit or not at all; you should just give the grader sufficient explanation of what should happen there.

26. Continuing problem 25, discuss briefly whether or not it would be preferable (a) to store the length of each string just before the first character, instead of using '128' just after the last character; or (b) to eliminate the extra '128' entirely and to save space by adding 128 to the final character.

27. J. H. Quick (a student) thought he spotted a bug in §671 and he was all set to collect \$40.96 because of programs like this:

```
\vbox{\moveright 1pt\hbox to 2pt{}
  \xleaders\lastbox\vskip 3pt}
```

(He noticed that TEX would give this vbox a width of 2 pt, and he thought that the correct width was 3 pt.) However, when he typed `\showlists` he saw that the leaders were simply

```
\xleaders 3.0
.\hbox(0.0+0.0)x2.0
```

and he noticed with regret the statement

$$\mathit{shift\_amount}(\mathit{cur\_box}) \leftarrow 0$$

in §1081.

Explain how §671 would have to be corrected, if the *shift_amount* of a leader box could be nonzero.

28. When your instructor made up this problem, he said '`\hbadness=-1`' so that TEX would print out the way each line of this paragraph was broken. (He sometimes wants to check line breaks without looking at actual output, when he's using a terminal that has no display capabilities.) It turned out that TEX typed this:

```
Loose \hbox (badness 0) in paragraph at lines 11--16
[]\tenrm When your instructor made up this problem, he said

Tight \hbox (badness 3) in paragraph at lines 11--16
\tenrm '\tentt \hbadness=-1\tenrm ' so that T[]X would print out the way each

Tight \hbox (badness 20) in paragraph at lines 11--16
\tenrm line of this paragraph was broken. (He sometimes wants to

Loose \hbox (badness 1) in paragraph at lines 11--16
\tenrm check line breaks without looking at actual output, when

Loose \hbox (badness 1) in paragraph at lines 11--16
\tenrm he's using a terminal that has no display capabilities.) It
```

Why wasn't anything shown for the last line of the paragraph?

29. How would the output of TEX look different if the *rebox* procedure were changed by deleting the statement '**if** $type(b) = vlist\_node$ **then** $b \leftarrow hpack(b, natural)$'? How would the output look different if the next conditional statement, '**if** $(is\_char\_node(p))$ ...' were deleted? (Note that box b might have been formed by *char_box*.)

30. What spacing does TEX insert between the characters when it typesets the formulas `$x==1$`, `$x++1$`, and `$x,,1$`? Find the places in the program where these spacing decisions are made.

31. When your instructor made up this problem, he said '`\tracingparagraphs=1`' so that his transcript file would explain why TEX has broken the paragraph into lines in a particular way. He also said '`\pretolerance=-1`' so that hyphenation would be tried immediately. The output is shown on the next page; use it to determine what line breaks would have been found by a simpler algorithm that breaks one line at a time. (The simpler algorithm finds the breakpoint that yields fewest demerits on the first line, then chooses it and starts over again.)

32. Play through the algorithms in parts 42 and 43, to figure out the contents of *trie_op*, *trie_char*, *trie_link*, *hyf_distance*, *hyf_num*, and *hyf_next* after the statement

```
\patterns{a1bc 2bcd3 ab1cd}
```

has been processed. Then execute the algorithm of §923, to see how TEX uses this efficient trie structure to set the values of *hyf* when the word `aabcd` is hyphenated. [The value of *hn* will be 5, and the values of $hc[1\,..\,5]$ will be $(96, 96, 97, 98, 99)$, respectively, when §923 begins.]

```
% This is the paragraph-trace output referred to in Problem 31:
[]\tenrm When your in-struc-tor made up this prob-lem, he
@ via @@0 b=0 p=0 d=100
@@1: line 1.2 t=100 -> @@0
said `\tentt \tracingparagraphs=1\tenrm ' so that his tran-script
@ via @@1 b=4 p=0 d=196
@@2: line 2.2 t=296 -> @@1
file would ex-plain why T[]X has bro-ken the para-
@\discretionary via @@2 b=175 p=50 d=46725
@@3: line 3.0- t=47021 -> @@2
graph
@ via @@2 b=25 p=0 d=1225
@@4: line 3.3 t=1521 -> @@2
into lines in a par-tic-u-lar way. He also said
@ via @@3 b=69 p=0 d=6241
@@5: line 4.1 t=53262 -> @@3
`\tentt \pretolerance=-1\tenrm ' so that hy-phen-ation would be
@ via @@5 b=43 p=0 d=2809
@@6: line 5.1 t=56071 -> @@5
tried im-me-di-ately. The out-put is shown on the next
@ via @@6 b=0 p=0 d=100
@@7: line 6.2 t=56171 -> @@6
page; use it to de-ter-mine what line breaks would
@ via @@7 b=153 p=0 d=36569
@@8: line 7.0 t=92740 -> @@7
have
@ via @@7 b=34 p=0 d=1936
@@9: line 7.3 t=58107 -> @@7
been found by a sim-pler al-go-rithm that breaks
@ via @@8 b=1 p=0 d=10121
@@10: line 8.2 t=102861 -> @@8
one
@ via @@9 b=15 p=0 d=10625
@@11: line 8.1 t=68732 -> @@9
line at a time. (The sim-pler al-go-rithm finds
@ via @@10 b=164 p=0 d=40276
@@12: line 9.0 t=143137 -> @@10
the
@ via @@10 b=0 p=0 d=100
@ via @@11 b=192 p=0 d=40804
@@13: line 9.0 t=109536 -> @@11
@@14: line 9.2 t=102961 -> @@10
break-point that yields fewest de-mer-its on the
@ via @@12 b=174 p=0 d=33856
@@15: line 10.0 t=176993 -> @@12
first
@ via @@12 b=41 p=0 d=12601
@ via @@13 b=75 p=0 d=7225
@ via @@14 b=75 p=0 d=7225
@@16: line 10.1 t=110186 -> @@14
line, then chooses it and starts over again.)
@\par via @@15 b=0 p=-10000 d=10100
@\par via @@16 b=0 p=-10000 d=100
@@17: line 11.2- t=110286 -> @@16
```

33. The *save_stack* is normally empty when a TeX program stops. But if, say, the user's input has an extra '{' (or a missing '}'), TeX will print the warning message

```
(\end occurred inside a group at level 1)
```

(see §1335).

Explain in detail how to change TeX so that such warning messages will be more explicit. For example, if the source program has an unmatched '{' on line 6 and an unmatched '\begingroup' on line 25, your modified TeX should give two warnings:

```
    (\end occurred when \begingroup
              on line 25 was incomplete)
    (\end occurred when { on line 6
              was incomplete)
```

You may assume that *simple_group* and *semi_simple_group* are the only group codes present on *save_stack* when §1335 is encountered; if other group codes are present, your program should call *confusion*.

34. (The following question is the most difficult yet most important of the entire collection. It was the main problem on the take-home final exam.)

The purpose of this problem is to extend TeX so that it will sell better in China and Japan. The extended program, called TeXX, allows each font to contain up to 65536 characters. Each extended character is represented by two values, its 'extension' x and its 'code' c, where both x and c lie between 0 and 255 inclusive. Characters with the same 'c' but different 'x' correspond to different graphics; but they have the same width, height, depth, and italic correction.

TeXX is identical to TeX except that it has one new primitive command: \xchar. If \xchar occurs in vertical mode, it begins a new paragraph; i.e., it's a ⟨horizontal command⟩ as on p. 283 of *The TeXbook*. If \xchar occurs in horizontal mode it should be followed by a ⟨number⟩ between 0 and 65535; this number can be converted to the form $256x + c$, where $0 \le x, c < 256$. The corresponding extended character from the current font will be appended to the current horizontal list, and the space factor will be set to 1000. (If $x = 0$, the effect of \xchar is something like the effect of \char, except that \xchar disables ligatures and kerns and it doesn't do anything special to the space factor. Moreover, no penalty is inserted after an \xchar that happens to be the \hyphenchar of the current font.) A word containing an extended character will not be hyphenated. The \xchar command should not occur in math mode.

Inside TeXX, an extended character (x, c) in font f is represented by two consecutive *char_node* items p and q, where we have $font(p) = null\_font$, $character(p) = qi(x)$, $link(p) = q$, $font(q) = f$, and $character(q) = qi(c)$. This two-word representation is used even when $x = 0$.

TeXX typesets an extended character by specifying character number $256x + c$ in the DVI file. (See the *set2* command in §585.)

If TeXX is run with the macros of plain TeX, and if the user types '\tracingall \xchar600 \showlists', the output of TeXX will include

```
{\xchar}
{horizontal mode: \xchar}
{\showlists}

### horizontal mode entered at line 0
\hbox(0.0+0.0)x20.0
\tenrm \xchar"258
spacefactor 1000
```

(since 600 is "258 in hexadecimal notation).

Your job is to explain in detail *all* changes to TeX that are necessary to convert it to TeXX.

[Note: A properly designed extension would also include the primitive operator \xchardef, analogous to \chardef and \mathdef, because a language should be 'orthogonally complete'. However, this additional extension has not been included as part of problem 34, because it presents no special difficulties. Anybody who can figure out how to implement \xchar can certainly also handle \xchardef.]

35. The first edition of *TeX: The Program* suggested that extended characters could be represented with the following convention: The first of two consecutive *char_node* items was to contain the font code and a character code from which the dimensions could be computed as usual; the second *char_node* was a *halfword* giving the actual character number to be typeset. Fonts were divided into two types, based on characteristics of their TFM headers; 'oriental' fonts always used this two-word representation, other fonts always used the one-word representation.

Explain why the method suggested in problem 34 is better than this. (There are at least two reasons.)

◇ Donald E. Knuth
Department of Computer Science
Stanford University
Stanford, CA 94305