Software

Answers to *Exercises for TEX: The Program*

Donald E. Knuth

Editor's note: The exercises apropos to these answers were printed in *TUGboat* 11, no. 2, pp. 165–170.

1. According to the index, *initialize* is declared in §4. It is preceded there by ⟨Global variables 13⟩, and §13 tells us that the final global variable appears in §1345. Turning to §1345, we find '*write_loc*: *pointer*;' and a comment. The comment doesn't get into the Pascal code. The mini-index at the bottom of page 535 tells us that '*pointer*' is a macro defined in §115. Our quest is nearly over, since §115 says that *pointer* expands to *halfword*, which is part of the Pascal program. Page ix tells us that lowercase letters of a `WEB` program become uppercase in the corresponding Pascal code; page x tells us that the underline in '*write_loc*' is discarded. Therefore we conclude that '`PROCEDURE INITIALIZE`' is immediately preceded in the Pascal program by '`WRITELOC:HALFWORD;`'.

But this isn't quite correct! The book doesn't tell the whole story. If we actually run `TANGLE` on `TEX.WEB` (without a change file), we find that '`PROCEDURE INITIALIZE`' is actually preceded by

```
{1345:}WRITELOC:HALFWORD;{:1345}
```

because `TANGLE` inserts comments to show the origin of each block of code.

2. The index tells us that *done5* and *done6* are never used. (They are included only for people who have to make system-dependent changes and/or extensions.)

3. Here we change the *input_ln* procedure of §31. One way is to replace the statements '$buffer[last] \leftarrow xord[f\uparrow]$; $get(f)$' by the following:

$$
\begin{array}{l}
\textbf{if } ord(f\uparrow) = \text{'}33 \textbf{ then}\\
\quad \textbf{begin } get(f);\\
\quad \textbf{if } (ord(f\uparrow) \ge \texttt{"@"}) \land (ord(f\uparrow) \le \texttt{"\_"}) \textbf{ then}\\
\qquad \textbf{begin } buffer[last] \leftarrow xord[chr(ord(f\uparrow) - \text{'}100)];\ get(f);\\
\qquad \textbf{end}\\
\quad \textbf{else } buffer[last] \leftarrow invalid\_code;\\
\quad \textbf{end}\\
\textbf{else begin } buffer[last] \leftarrow xord[f\uparrow];\ get(f);\\
\quad \textbf{end};
\end{array}
$$

4. The new string essentially substitutes "quarters" `q` (of value 25) for "dimes" `x` (of value 10). Playing through the code of §69 tells us that 69 is now represented by `lvvviv` and 9999 is `mmmmmmmmmcmqcvqiv`. (The first nine `m`'s make 9000; then `cm` makes 900; then `qc` makes 75; then `vq` makes 20; and `iv` makes the remaining 4.)

5. Because it may be decreased by 1 in §1293 before being increased by 1 in §82. (The code in §1293 decreases *error_count* because "showing" uses the *error* subroutine although it isn't really an error.)

6. The `q` becomes `Q` in §83. This causes §86 to print '`OK, entering \batchmode`', after which *selector* is decreased so that '`...`' and ⟨return⟩ are *not* printed on the terminal! (They appear only in the log file, if it has been opened.) This is TEX's way of confirming that `\batchmode` has indeed been entered.

7. (a) Arithmetic overflow might occur when computing $t*297$, because $7230585 \times 297 = 2^{31} + 97$. (b) Some sort of test is need to avoid division by zero when $0 < s < 297$. If $s < 1663497$ then s **div** $297 < 5601$, and $7230585/5600$ is a bit larger than 1291 so we will have $r > 1290$ in such a case. The threshold value has therefore been chosen to save division whenever possible. (One student suggested that the statement '$r \leftarrow t$' be replaced by '$r \leftarrow 1291$'. That might or might not be faster, depending on the computer and the Pascal compiler. In machine language one would '**goto**' the statement that sets $badness \leftarrow inf\_bad$, but that is inadmissible Pascal.) (c) If we get to §128 with $r = p + 1$, we will try to make a node of size 1, but then there's no room for the *node_size* field. (d) If we get to §129 with only one node available, we'll lose everything and *rover* will be invalid. (Older versions of TEX have a more complicated test in §127, which would suppress going to §129 if there were two nodes available. That was unnecessarily cautious.) (e) This is a subtle one. The lower part of memory must not be allowed to grow so large that a *node_size* value could ever exceed *max_halfword* when nodes are being merged together in §127.

8. We assume that *min_quarterword* = *min_halfword* = 0.

100:	0	0	0	*type* (*hlist_node*), , *link*
101:			6553600	*width* (100 pt)
102:			0	*depth*
103:			655360	*height* (10 pt)
104:			0	*shift_amount*
105:	1	2	200	*glue_sign* (*stretching*), *glue_order* (*fill*), *list_ptr*
106:		10.0		*glue_set* (type *real*)
200:	7	1	10003	*type* (*disc_node*), *replace_count*, *link*
201:		300	10000	*pre_break*, *post_break*
300:	11	1	0	*type* (*kern_node*), *subtype* (*explicit*), *link*
301:			655360	*width* (10 pt)
400:	6	0	0	*type* (*ligature_node*), , *link*
401:	1	11	10001	*font*, *character*, *lig_ptr*
500:	12	0	600	*type* (*penalty_node*), , *link*
501:			5000	*penalty*
600:	10	0	700	*type* (*glue_node*), *subtype* (*normal*), *link*
601:		8	0	*glue_ptr* (*fill_glue*), *leader_ptr*
700:	1	0	0	*type* (*vlist_node*), , *link*
701:			655360	*width* (10 pt)
702:			32768	*depth* (0.5 pt)
703:			327680	*height* (5 pt)
704:			-327680	*shift_amount* (−5 pt)
705:	0	0	800	*glue_sign* (*normal*), *glue_order* (*normal*), *list_ptr*
706:		0.0		*glue_set* (type *real*)
800:	0	0	900	*type* (*hlist_node*), , *link*
801:			655360	*width* (10 pt)
802:			0	*depth*
803:			327680	*height* (5 pt)
804:			0	*shift_amount*
805:	0	0	10004	*glue_sign* (*normal*), *glue_order* (*normal*), *list_ptr*
806:		0.0		*glue_set* (type *real*)
900:	2	0	0	*type* (*rule_node*), , *link*
901:			-1073741824	*width* (*null_flag*)
902:			0	*depth*
903:			32768	*height* (0.5 pt)
10000:	1	"U"	400	*font*, *character*, *link*
10001:	1	"f"	10002	*font*, *character*, *link*
10002:	1	"f"	0	*font*, *character*, *link*
10003:	1	"!"	500	*font*, *character*, *link*
10004:	2	"d"	10005	*font*, *character*, *link*
10005:	2	"a"	0	*font*, *character*, *link*

9. (Norwegian Americans will recognize this as an 'Uff da' joke.) The output of *short_display* is

```
\large Uff []
```

since *short_display* shows the pre-break and post-break parts of a discretionary (but not the replacement text). However, if this box were output by *hlist_out*, the discretionary break would not be effective; the result would be a box 100 pt wide, beginning with a large '!' and ending with a small 'da', the latter being raised 5 pt and underlined with a 0.5 pt-rule.

10. Since *prev_depth* is initially *ignore_depth*, we get

```
### vertical mode entered at line 1
                         (\output routine)
prevdepth -999.99998, prevgraf 1 line
```

11. According to §236, *int_base* + 17 is where *mag* is stored. (One of the definitions suppressed by an ellipsis on page 101 is *mag*; you can verify this by checking the index!) The initial value of *mag* is set in §240. Hence *show_eqtb* branches to §242 and prints '`\mag=1000`'.

12. In the following chart, '(3)' means a value at level three, and '—' is a level boundary:

															(2) 9			
										(1) 6	(1) 6			—	—			
								—	—	—	—		(1) 8	(1) 8	(1) 8	(1) 8	(1) 8	
							(1) 4	(1) 4	(1) 4	(1) 4	(1) 4	(1) 4	(1) 4	(1) 4	(1) 4	(1) 4	(1) 4	
					(1) 2	(1) 2	(1) 2	(1) 2	(1) 2	(1) 2	(1) 2	(1) 2	(1) 2	(1) 2	(1) 2	(1) 2	(1) 2	
save_stack:				—	—	—	—	—	—	—	—	—	—	—	—	—	—	
xeq_level[*p*]:	(1)	(1)	(1)	(1)	(2)	(1)	(2)	(2)	(1)	(3)	(1)	(1)	(2)	(2)	(3)	(2)	(2)	(1)
eqtb[*p*].*int*:	0	1	2	2	3	4	5	5	6	7	8	8	9	9	10	9	10	8
operations:	`\day=0`	`\g`	`\a`	`{`	`\a`	`\g`	`\a`	`{`	`\g`	`\a`	`\g`	`}`	`\a`	`{`	`\a`	`}`	`\a`	`}`

The final value is therefore `\day=8`.

13. (reference count), *match* `!`, *match* `#`, *left_brace* `[`, *end_match*, *left_brace* `{`, *mac_param* `#`, *right_brace* `]`, *mac_param* `!`, *out_param* 2, *left_brace* `[`. Notice that the *left_brace* before the *end_match* is repeated at the end of the replacement text, because it has been matched (and therefore removed from the input).

14. According to §233, *show_eqtb*(*every_par_loc*) calls *show_token_list* with the limit $l = 32$. According to §292, we want the token list to contain a token that prints as many characters as possible when $tally = 31$; the value of *tally* is increased on every call to *print_char* (§58). By studying the cases in §294, we conclude that the worst case occurs when a *mac_param* is printed, and when the character c actually prints as three characters. The statement '*print_esc*("`ETC.`")' in §292 will print seven additional characters if the current *escape_char* is another tripler. (Longer examples are possible only if TeX has a bug that tweaks one of the outputs '`\CLOBBERED.`' or '`\BAD.`' in §293; but this can't happen.)

In other words, a worst-case example such as

```
\escapechar=`\^^M \catcode`\^^I=6
\everypar{1234567890123456789012345678901^^Ietc.}
```

in connection with the suggested test line will print

```
{restoring ^^Meverypar=1234567890123456789012345678901^^I^^I^^METC.}
```

thereby proving that 44 characters can be printed by *show_eqtb*(*every_par_loc*).

15. Here we must look at the *get_next* procedure, which scans the *buffer* in strange ways when two identical characters of category 7 (*sup_mark*) are found. After the `\catcode` of open-quote has been set to 7, *get_next* begins to scan a control sequence in §354, which goes to §355 and finds a space after `‘‘`. Since a space is code *'40*, it is changed to *'140*, and the buffer contents are shifted left 2. By strange coincidence, *'140* is again an open-quote character, so we get back to §355, which changes `‘‘(` to `h` and goes back to *start_cs* a third time. Now we go to §356 and then back to §355 and *start_cs*, having changed `‘‘)` to `i`. The fourth round, similarly, changes `‘‘‘` to a blank space, and the fifth round finishes the control sequence.

If we try to input the stated line, `INITEX` will come to a halt as follows:

```
! Undefined control sequence.
<*> \catcode‘‘=7 \hi
                    !\error
```

This proves that the *buffer* now says `\hi !`.

16. The error message in question is

```
! Undefined control sequence.
<*> \endlinechar=‘! \error
                          ^^M
```

and our job is to explain the appearance of `^^M`. The standard `\endlinechar` is *carriage_return*, according to §240; this is *'15* according to §22, and *'15* is `^^M` in ASCII code. Thus, a *carriage_return* is normally placed at the end of each line when it's read into the *buffer* (see §360). This *carriage_return* is not usually printed in an error message, because it equals the *end_line_char* (see §318). We see it now because *end_line_char* has changed.

Incidentally, if the input line had been

```
\endlinechar=‘!\error
```

(without the space after the `!`), we wouldn't have seen the `^^M`. Why not? Because TeX calls *get_next* when looking for the optional space after the ASCII constant `‘!` (see §442–443), hence the undefined control sequence `\error` is encountered before *end_line_char* has been changed!

17. One problem is to figure out which control sequence is undefined; it seems to be the '?', since this character has been made active. One clue is to observe from §312 and §314 that '`<recently read>`' can be printed only when $base\_ptr = input\_ptr$, $state = token\_list$, $token\_type = backed\_up$, and $loc = null$. A token list of type *backed_up* usually contains only a single item; in that case, the control sequence name must be '`How did this happen?`', and we have a problem getting an active character into a control sequence name.

But an arbitrarily long token list of type *backed_up* can be created with the `\lowercase` operation (see §1288). In that case, however, the right brace that closes `\lowercase` is almost always still present in TeX's input state, and it would show up on the error message. (The *back_list* procedure of §323 does not clear a completed token list off of the stack.) We have to make TeX clear off its stack before the `}` is scanned.

At this point the exercise begins to resemble "retrograde chess" problems. Here is one solution; since it requires a very long input line, it has been broken into a three-line answer:

```
\def\answer{\let~\expandafter\lccode‘!=‘H% [line has been broken]
 ~\lowercase~{~!~o~w~ ~d~i~d~ ~t~h~i~s~  % [line has been broken]
 ~h~a~p~p~e~n~?}}
```

(The '`H`' is a lowercase '`!`'; a chain of `\expandafter`s is used to make the right brace disappear from the stack.)

Another approach uses `\csname`, and manufactures a `?` from a `!`:

```
\def\answer{\def\a##1{{\global\let##1?\aftergroup##1}}% [broken]
 \escapechar‘H\lccode‘!|/‘?                            % [broken]
 \lowercase{\expandafter\a\csname ow did this happen!\endcsname}}
```

But there is a (devious) one-line solution, which makes the invisible *carriage_return* following `\answer` into a right brace:

```
\def\answer{\catcode13=2\lccode`!=H\lowercase\bgroup!ow did this happen?}
```

18. (The answer to this problem was much more difficult to explain in class than I had thought it would be, so I guess it was also much more difficult for the students to solve than I had thought it would be. After my first attempt to explain the answer, I decided to make up a special version of TeX that would help to clarify the scanning routines. This special program, called DemoTeX, is just like ordinary TeX except that if `\tracingstats>2` the user is able to watch TeX's syntax routines in slow motion. The changes that convert TeX to DemoTeX are explained in the appendix below. Given DemoTeX, we tried a lot of simple examples of things like '`\hfuzz=1.5pt`' and '`\catcode`a=11`' before plunging into exercise 18 in which everything happens at once. While we were discussing input stacks, by the way, we found it helpful to consider the behavior of TeX on the following input:

```
\output{\botmark}
\def\a{\error}
\mark{
 \everyvbox{
  \everypar{
   \everydisplay{
    \everyhbox{
     \everymath{\noexpand\a}
     $\relax}
    \hbox\bgroup\relax}
   $$\relax}
  \noindent\relax}
 \vbox\bgroup\relax}
\hbox{}\vfill\penalty-10000
```

Here `\penalty` triggers `\botmark`, which defines `\everyvbox` and begins a `\vbox`, which defines `\everypar` and begins a `\par`, which defines `\everydisplay` and begins a `\display`, etc.)

The first line is essentially

```
\gdef\a#1d#2#3{#2}
```

where the second 'd' has catcode 12 (*other_char*). Hence the second `d` will match a `d` that is generated by `\romannumeral`. In this line, *scan_int* is called only to scan the `` `d `` and the `12`.

The second line calls *scan_dimen* in order to evaluate the right-hand side of the assignment to `\hfuzz`. After *scan_dimen* has used *scan_int* to read the '`100`', it calls *scan_keyword* in order to figure out the units. But before the units are known to be '`pt`' or '`pc`', an `\ifdim` must be expanded. Here we need to call *scan_dimen* recursively, twice; it finds the value 12 pt on the left-hand side, and is interrupted again while *scan_keyword* is trying to figure out the units on the right-hand side. Now a chain of `\expandafter`s causes `\romannumeral888` to be expanded into `dccclxxxviii`, and then we have to parse `\a dccclxxxviii`. Here `#1` will be `\else`, `#2` and `#3` will each be `c`; the expansion therefore reduces to `cclxxxviii\relax\fi`. The first '`c`' completes the second '`Pc`', and the `\ifdim` test is true. Therefore the second '`c`' can complete the first '`Pc`', and `\hfuzz` is set equal to 1200 pt. The characters `lxxxviii` now begin a paragraph. The `\fi` takes the `\ifdim` out of TeX's condition stack.

(The appendix below gives further information. Examples like this give some glimmering of the weird maneuvers that can be found in the `TRIP` test, an intricate pattern of unlikely code that is used to validate all implementations of TeX.)

19. If, for example, `\thickmuskip` has the value `5mu plus 5mu` that plain TeX gives it, the first command changes its value to `-5mu plus -5mu`, because *scan_glue* in §461 will call *scan_something_internal* with the second argument *true*; this will cause all three components of the glue to be negated (see §431).

The second command, on the other hand, tells TeX to expand '`\the\thickmuskip`' into a sequence of characters, so it is equivalent to

```
\thickmuskip=-5mu plus 5mu
```

(The minus sign doesn't carry into the stretch component of glue, since §461 applies *negate* only to the first dimension found.)

This problem points out a well-known danger that is present in any text-macro-expanding system.

20. We'd have a funny result that two macro texts would be considered to match by `\ifx` unless the first one (the one starting at q when we begin §508) is a proper prefix of the second. (Notice the statement '$p \leftarrow null$' inside the **while** loop.)

21. Because the byte in $dvi\_buf[dvi\_ptr - 1]$ is usually not an operation code, and it just might happen to equal *push*.

22. $2_y\,7_d\,1_d\,8_z\,2_y\,8_z\,1_d\,8_z\,2_y\,8_z\,4_y\,5_z\,9_d\,0_d\,4_y\,5_z$.

23. TEX is in 'no mode' only while processing `\write` statements, and the mode is printed during `\write` only when *tracing_commands* > 1 during *expand*. We might think that `\catcode` operations are necessary, so that the left and right braces for `\write` exist; but it's possible to let TEX's error-recovery mechanism supply them! Therefore the shortest program that meets the requirements is probably the following one based on an idea due to Ronaldo Amá, who suggests putting

```
\batchmode\tracingcommands2
  \immediate\write!\nomode
```

into a file. (Seven tokens total.)

24. When *error* calls *get_token*, because the user has asked for tokens to be deleted (see §88), a second level of *error* is possible, but further deletions are disallowed (see §336 and §346). However, insertions are still allowed, and this can lead to a third level of *error* when *overflow* calls *succumb*.

For example, let's assume that *max_in_open* $=$ 6. Then you can type '`\catcode`?=15 \x`' and respond to the undefined control sequence error by saying '`i\x??`' six times. This leads to a call of *error* in which six '`<insert>`' levels appear; hence *in_open* $= 6$, and one more insertion will be the last straw. At this point, type '`1`'; this enters *error* at a second level, from which '`i`' will enter *error* a third time. (The run-time stack now has *main_control* calling *get_x_token* calling *expand* calling *error* calling *get_token* calling *get_next* calling *error* calling *begin_file_reading* calling *overflow* calling *error*.)

25. In §38, define *str_number* to be the same as *pool_pointer*, and define *str_end* $= 128$. In §39, delete the declaration of *str_start*. In §40, declare

> **function** *length*(*s* : *str_number*): *integer*;
>   **var** *t*: *pool_pointer*;
>   **begin** $t \leftarrow s$;
>   **while** $str\_pool[t] \neq str\_end$ **do** *incr*(*t*);
> $length \leftarrow t - s$;
>   **end**;

In §41, define $cur\_length \equiv (pool\_ptr - str\_ptr)$. In §43, declare

> **function** *make_string*: *str_number*; { current string enters the pool }
>   **var** *t*: *str_number*; { the result }
>   **begin** *str_room*(1); *append*(*str_end*);
> $t \leftarrow str\_ptr$; $str\_ptr \leftarrow pool\_ptr$; $make\_string \leftarrow t$;
>   **end**;

In §44, we can

> **define** $flush\_string \equiv$ **begin repeat** *decr*(*str_ptr*);
>   **until** $str\_pool[str\_ptr - 1] = str\_end$;
> $pool\_ptr \leftarrow str\_ptr$;
>   **end**

The comparison function in §45 is used only in §259, where we can replace

'**if** $length(text(p)) = l$ **then if** $str\_eq\_buf(text(p), j)$'

by '**if** $str\_eq\_buf(text(p), j, l)$'. The function now has three parameters:

```
function str_eq_buf(s : str_number; k, l : integer): boolean;
        { test equality of strings }
  label exit;
  var j: pool_pointer;   { running index }
  begin j ← s; s ← s + l;
  if str_pool[s] ≠ str_end then str_eq_buf ← false
  else begin while j < s do
      begin if str_pool[j] ≠ buffer[k] then
        begin str_eq_buf ← false; return; end;
      incr(j); incr(k);
      end;
    str_eq_buf ← true;
    end;
exit: end;
```

The procedure of §46 is modified in an obvious, similar way.

The first three statements of §47 become just two: '*pool_ptr* ← 128; *str_ptr* ← 128'. The body of the **for** loop in §48 becomes just

```
if (⟨Character k cannot be printed 49⟩) then
  if k < '100 then str_pool[k] ← k + '100
  else str_pool[k] ← k − '100
else str_pool[k] ← k
```

In §59, variable j is no longer needed. If $0 \le s < 128$ and if s isn't the current new-line character, we now say

```
begin if str_pool[s] ≠ s then
  begin print_char("^"); print_char("^");
  end;
print_char(str_pool[s]);
end
```

In the other case, where $s \ge 128$, we say

```
while str_pool[s] ≠ str_end do
  begin print_char(str_pool[s]); incr(s);
  end
```

In §407, similarly, variable k is eliminated; the loop on k becomes a loop on s, **while** *str_pool*[s] ≠ *str_end*.

In §464, replace the two occurrences of '*str_start*[*str_ptr*]' by '*str_ptr*'.

The first loop in §603 becomes

```
k ← font_area[f];
while str_pool[k] ≠ str_end do
  begin dvi_out(str_pool[k]); incr(k);
  end
```

and the second is like unto it.

26. Let's assume that we have a machine in which *str_pool* is addressed by byte number, so that 8-bit values take no more space than 7-bit values. Method (a) requires us to impose a limit on the length of strings: 255 characters max. This isn't unreasonable, because the only important use of longer strings is in the implementation of `\special`, when the restriction doesn't actually apply (since §1368 doesn't call *make_string*). But method (a) saves no space and little or no time by comparison with the simpler method of problem 25. Problem 25 saves about one byte per string, compared to the text's way. Method (b) saves another byte per string but at the expense of considerable programming complexity; it requires awkward special-casing to deal with empty strings.

27. We'd replace '*width*(g)' by

$$width(g) + shift\_amount(g)$$

(twice). Similar changes would be needed in §656. (But a box shouldn't be able to retain its *shift_amount*; this quantity is a property of the list the box is in, not a property of the box itself.)

28. The final line has infinite stretchability, since plain TeX sets `\parfillskip=0pt plus 1fil`. Reports of loose, tight, underfull, or overfull boxes are never made unless $o = normal$ in §658 and §664.

29. If a vbox is repackaged as an hbox, we get really weird results because things that were supposed to stack up vertically are placed together horizontally. The second change would be a lot less visible, except in characters like V where there is a large italic correction; the character would be centered without taking its italic correction into account. (The italic correction in math mode is the difference between horizontal placement of superscripts and subscripts in formulas like V_2^2.)

30. The spacing can be found by saying

`$x==1$ $x++1$ $x,,1$ \tracingall\showlists.`

Most of the decisions are made in §766, using the spacing table of §764. But the situation is trickier in the case of `+`, because a *bin_noad* must be preceded and followed by a noad of a suitable class. In the formula `$x++1$`, the second `+` is changed from *bin_noad* to *ord_noad* in §728. It turns out that thick spaces are inserted after the x and before the 1 in '$x == 1$'; medium spaces are inserted before each + sign in '$x + +1$'; thin spaces are inserted after each comma in '$x, , 1$'.

31. The behavior of the simpler algorithm, which we may call Brand X, can be deduced from the demerits values ('`d=`') in the trace output. There is only one reasonable choice, `@@1`, for the first line; and there's only one, `@@2`, for the second. But for the third, a line from `@@2` to `@@3` (the break after '`para-`') has 46725 demerits, which certainly looks worse than the 1225 demerits from `@@2` to `@@4`. This, however, leads Brand X into a trap, since there's no good way to continue from `@@4`. Similarly, Brand X will choose to go from `@@7` to `@@9`, and this forces it to `@@11` and then infelicitously to `@@13` (because the syllable '`break-`' is too long to be squeezed in). The resulting paragraph, as typeset by Brand X, looks like this (awful):

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. (This exercise takes awhile, but the data structures are especially interesting; the hyphenation algorithm is a nice little part of the program that can be studied in isolation.) The following tables are constructed:

	op	*char*	*link*
trie[96]	0	96	1
trie[97]	0	97	5
trie[98]	0	97	2
trie[100]	1	98	3
trie[102]	1	99	4
trie[103]	0	98	6
trie[105]	3	99	4

	[1]	[2]	[3]
hyf_distance	2	0	3
hyf_num	1	3	2
hyf_next	0	0	2

Given the word `aabcd`, it is interesting to watch §923 produce the hyphenation numbers '${}_0\mathrm{a}_0\mathrm{a}_2\mathrm{b}_1\mathrm{c}_0\mathrm{d}_3$' from this trie.

33. The idea is to keep line numbers on the save stack. Scott Douglass has observed that, although TeX is careful to keep *cur_boundary* up to date, nothing important is ever done with it; hence the *save_index* field in level-boundary words is not needed, and we have an extra halfword to play with! (The present data structure has fossilized elements left over from old incarnations of TeX.) However, line numbers might get larger than a halfword; it seems better to store them as fullword integers.

This problem requires changes to three parts of the program. First, we can extend §1063 as follows:

⟨Cases of *main_control* that build boxes and lists 1056⟩ +≡
non_math(*left_brace*): **begin** *saved*(0) ← *line*; *incr*(*save_ptr*); *new_save_level*(*simple_group*);
  end; {the line number is saved for possible use in warning message}
any_mode(*begin_group*): **begin** *saved*(0) ← *line*; *incr*(*save_ptr*); *new_save_level*(*semi_simple_group*);
  end;
any_mode(*end_group*): **if** *cur_group* = *semi_simple_group* **then**
    begin *unsave*; *decr*(*save_ptr*); {pop unused line number from stack}
    end
  else *off_save*;

A similar change is needed in §1068, where the first case becomes

```
simple_group: begin unsave; decr(save_ptr);  { pop unused line number from stack }
  end;
```

Finally, we replace lines 6–11 of §1335 by code for the desired messages:

```
while cur_level > level_one do
  begin print_nl("("); print_esc("end␣occurred␣when␣");
  case cur_group of
  simple_group: print_char("{");
  semi_simple_group: print_esc("begingroup");
  othercases confusion("endgroup")
  endcases;
  print("␣on␣line␣"); unsave; decr(save_ptr); print_int(saved(0)); print("␣was␣incomplete)");
  end;
while cond_ptr ≠ null do
  begin print_nl("("); print_esc("end␣occurred␣when␣"); print_cmd_chr(if_test, cur_if);
```

34. First, §2 gets a new paragraph explaining what TEXX is, and the banner line changes:

```
define banner ≡ ´This␣is␣TeXX,␣Version␣2.2´  { printed when TEX starts }
```

Then we add two new definitions in §134:

```
define is_xchar_node(#) ≡ (font(#) = font_base)  { is this char_node extended? }
define bypass_xchar(#) ≡
        if is_xchar_node(#) then # ← link(#)
```

(It's necessary to say *font_base* here instead of *null_font*, because *null_font* isn't defined until later.)

The *short_display* routine of §174 can treat an `\xchar` like an ordinary character, because *print_ASCII* makes no restrictions. Here is one way to handle the change:

```
procedure short_display(p : integer);  { prints highlights of list p }
  label done;
  var n: integer;  { for replacement counts }
    ext: integer;  { amount added to character code by xchar }
  begin ext ← 0;
  while p > mem_min do
    begin if is_char_node(p) then
      begin if p ≤ mem_end then
        begin if is_xchar_node(p) then
          begin ext ← 256 * (qo(character(p))); goto done;
          end;
        if font(p) ≠ font_in_short_display then
          begin if (font(p) < font_base) ∨ (font(p) > font_max) then print_char("*")
          else ⟨Print the font identifier for font(p) 267⟩;
          print_char("␣"); font_in_short_display ← font(p);
          end;
```

```
        print_ASCII(ext + qo(character(p)));   ext ← 0;
        end;
      end
    else ⟨Print a short indication of the contents of node p 175⟩;
  done: p ← link(p);
    end;
  end;
```

A somewhat similar change applies in §176:

```
procedure print_font_and_char(p : integer);   { prints char_node data }
  label reswitch;
  var ext: integer;   { amount added to character code by xchar, or −1 }
  begin ext ← −1;
reswitch: if p > mem_end then print_esc("CLOBBERED.")
  else begin if is_xchar_node(p) then
      begin ext ← qo(character(p)); p ← link(p); goto reswitch; end;
    if (font(p) < font_base) ∨ (font(p) > font_max) then print_char("*")
    else ⟨Print the font identifier for font(p) 267⟩;
    print_char("␣");
    if ext < 0 then print_ASCII(qo(character(p)))
    else begin print_esc("xchar"); print_hex(ext * 256 + qo(character(p)));
      end;
    end;
  end;
```

(These routines must be extra-robust.) The first line of code in §183 now becomes

```
if is_char_node(p) then
  begin print_font_and_char(p);
  bypass_xchar(p);
  end
```

In §208 we introduce a new operation code,

```
define xchar_num = 17
          { extended character ( \xchar ) }
```

Every opcode that follows it in §208 and §209, from *math_char_num* to *max_command*, must be increased by 1. We also add the following lines to §265 and §266, respectively:

```
    primitive("xchar", xchar_num, 0);
  xchar_num: print_esc("xchar");
```

This puts the new command into TEXX's repertoire.

The next thing we need to worry about is what to do when \xchar occurs in the input. It's convenient to add a companion procedure to *scan_char_num* in §435:

```
procedure scan_xchar_num;
  begin scan_int;
  if (cur_val < 0) ∨ (cur_val > 65535) then
    begin print_err("Bad␣character␣code");
    help2("An␣\xchar␣number␣must␣be␣between␣0␣and␣255.")
    ("I␣changed␣this␣one␣to␣zero."); int_error(cur_val); cur_val ← 0;
    end;
  end;
```

Similarly, *new_character* gets a companion in §582:

> **function** *new_xchar*(*f* : *internal_font_number*; *c* : *integer*): *pointer*;
> **var** *p*, *q*: *pointer*; { newly allocated nodes }
> **begin** $q \leftarrow$ *new_character*(*f*, *c* **mod** 256);
> **if** $q =$ *null* **then** *new_xchar* $\leftarrow$ *null*
> **else begin** $p \leftarrow$ *get_avail*; *font*(*p*) $\leftarrow$ *font_base*; *character*(*p*) $\leftarrow$ *qi*((*c* **div** 256)); *link*(*p*) $\leftarrow$ *q*;
> *new_xchar* $\leftarrow$ *p*;
> **end**;
> **end**;

Extended characters can be output properly if we replace the opening lines of the code in §620 by these:

> *reswitch*: **if** *is_char_node*(*p*) **then**
> **begin** *synch_h*; *synch_v*;
> **repeat if** *is_xchar_node*(*p*) **then**
> **begin** $f \leftarrow$ *font*(*link*(*p*));
> **if** *character*(*p*) $=$ *qi*(0) **then** $p \leftarrow$ *link*(*p*); { bypass zero extension }
> **end**
> **else** $f \leftarrow$ *font*(*p*);
> $c \leftarrow$ *character*(*p*);
> **if** $f \neq$ *dvi_f* **then** ⟨ Change font *dvi_f* to *f* 621 ⟩;
> **if** *is_xchar_node*(*p*) **then**
> **begin** *dvi_out*(*set1* + 1); *dvi_out*(*qo*(*c*)); $p \leftarrow$ *link*(*p*); $c \leftarrow$ *character*(*p*);
> **end**
> **else if** $c \geq$ *qi*(128) **then** *dvi_out*(*set1*);
> *dvi_out*(*qo*(*c*));

Many of the processing routines include a statement of the form '$f \leftarrow$ *font*(#)', which we want to do only after bypassing the first half of an extended character. This can be done by inserting the following statements:

bypass_xchar(*p*)	in §654;
bypass_xchar(*s*)	in §842;
bypass_xchar(*cur_p*)	in §867;
bypass_xchar(*s*)	in §871;
bypass_xchar(*p*)	in §1147.

In §841 we need to do a little more than a simple bypass:

> **if** *is_char_node*(*v*) **then**
> **begin if** *is_xchar_node*(*v*) **then**
> **begin** $v \leftarrow$ *link*(*v*); *decr*(*t*);
> { an xchar counts as two chars }
> **end**;

Two changes are needed in order to suppress hyphenation in words that contain extended characters. First we insert

> **if** *hf* $=$ *font_base* **then goto** *done1*;
> { *is_xchar_node*(*s*) }

after the third line of §896. Then we replace '**endcases**;' in §899 by

> **endcases**
> **else if** *is_xchar_node*(*s*) **then goto** *done1*;

If `\xchar` appears in math mode, we want to recover from the error by including *mmode* + *xchar_num* in the list of cases in §1046. If `\xchar` appears in vertical mode, we want to begin a paragraph by including *vmode* + *xchar_num* in the second list of cases in §1090.

But what if `\xchar` appears in horizontal mode? To handle this, we might as well rewrite §1122:

1122. We need only two more things to complete the horizontal mode routines, namely the `\xchar` and `\accent` primitives.

⟨Cases of *main_control* that build boxes and lists 1056⟩ +≡
hmode + *xchar_num*: **begin** *scan_xchar_num*; *link*(*tail*) ← *new_xchar*(*cur_font*, *cur_val*);
  if *link*(*tail*) ≠ *null* **then** *tail* ← *link*(*link*(*tail*));
  space_factor ← 1000;
  end;
hmode + *accent*: *make_accent*;

Finally, we need to extend *make_accent* so that extended characters can be accented. (Problem 34 didn't call for this explicitly, but TeXX should surely do it.) This means adding a new case in §1124:

else if *cur_cmd* = *xchar_num* **then**
  begin *scan_xchar_num*; *q* ← *new_xchar*(*f*, *cur_val*);
  end

and making changes at the beginning and end of §1125:

⟨Append the accent with appropriate kerns, then set *p* ← *q* 1125⟩ ≡
begin *t* ← *slant*(*f*)/*float_constant*(65536);
if *is_xchar_node*(*q*) **then** *i* ← *char_info*(*f*)(*character*(*link*(*q*)))
else *i* ← *char_info*(*f*)(*character*(*q*));
w ← *char_width*(*f*)(*i*);
⋮
subtype(*tail*) ← *acc_kern*; *link*(*p*) ← *tail*;
if *is_xchar_node*(*q*) **then** { in this case we want to bypass the xchar part }
  begin *tail_append*(*q*); *p* ← *link*(*q*);
  end
else *p* ← *q*;
end

35. The main reason for preferring the method of problem 34 is that the italic correction operation (§1113) would be extremely difficult with the other scheme. Other advantages are: (a) Division by 256 is needed only once; TeXX's main loops remain fast. (b) Comparatively few changes from TeX itself are needed, hence other ripoffs of TeX can easily incorporate the same ideas. (c) Since fonts don't need to be segregated into 'oriental' and 'occidental', `\xchar` has wide applicability. For example, it gives users a way to suppress ligatures and kerns; it allows large fonts to have efficient 256-character subsets of commonly-used characters. (d) The conventions of TeXX match those of the GF files produced by METAFONT.

The only disadvantage of the TeXX method is that it requires all characters whose codes differ by multiples of 256 to have the same box size. But this is a minor consideration.

Appendix

The solution to problem 18 refers to a special version of TeX called DemoTeX, which allows users to see more details of the scanning process. DemoTeX is formed by making a few changes to parts 24–26 of TeX.

First, in §341, the following code is placed between '*exit*:' and '**end**':

if *tracing_stats* > 2 **then**
  begin *k* ← *trace_depth*; *print_nl*("");
  while *k* > 0 **do**
    begin *print*("␣"); *decr*(*k*);
    end;
  print("|"); *print_char*("␣");
  if *cur_cs* > 0 **then**
    begin *print_cs*(*cur_cs*);
    print_char("=");
    end;
  print_cmd_chr(*cur_cmd*, *cur_chr*);
  end;

(A new global variable, *trace_depth*, is declared somewhere and initialized to zero. It is used to indent the output of DemoTeX so that the depth of subroutine nesting is displayed.)

At the beginning of *expand* (in §366), we put the statements

> *incr*(*trace_depth*);
> **if** *tracing_stats* > 2 **then** *print*("␣<x");

this prints '<x' when *expand* begins to expand something. The same statements are inserted at the beginning of *scan_int* (§400), *scan_dimen* (§448), and *scan_glue* (sec461), except that *scan_int* prints '<i', *scan_dimen* prints '<d', and *scan_glue* prints '<g'. (Get it?) We also insert complementary code at the end of each of these procedures:

> *decr*(*trace_depth*);
> **if** *tracing_stats* > 2 **then** *print_char*(">");

this makes it clear when each part of the scanner has done its work.

Finally, *scan_keyword* is instrumented in a similar way, but with explicit information about what keyword it is seeking. The code

> *incr*(*trace_depth*);
> **if** *tracing_stats* > 2 **then**
>   **begin** *print*("␣<´"); *print*(*s*);
>   *print_char*("´");
>   **end**;

is inserted at the beginning of §407, and

> **if** *tracing_stats* > 2 **then** *print_char*("*");
> *exit*: *decr*(*trace_depth*);
> **if** *tracing_stats* > 2 **then** *print_char*(">");
> **end**;

replaces the code at the end. (Here '*' denotes 'success': the keyword was found.)

For example, here's the beginning of what DemoTeX prints out when scanning the right-hand side of the assignment to \hfuzz in problem 18:

```
|! the character = <d
 |! the character 1 <i
  |! the character 1
  |! the character 0
  |! the character 0
  |! the letter P>
 |! the letter P <'em'
  |! the letter P> <'ex'
  |! the letter P> <'true'
  |! the letter P> <'pt'
  |! the letter P
  |! \ifdim =\ifdim <x <d
    |! the character 1 <i
     |! the character 1
     |! the character 2
     |! the letter p>
    |! the letter p <'em'
     |! the letter p> <'ex'
     |! the letter p> <'true'
     |! the letter p> <'pt'
     |! the letter p
     |! the letter t*>
    |! the character =>
```

(After seeing '=', TeX calls *scan_dimen*. The next character seen is '1'; *scan_dimen* puts it back to be read again and calls *scan_int*, which finds '100', etc. This output demonstrates the fact that TeX frequently uses *back_input* to reread a character, when it isn't quite ready to deal with that character.)

Acknowledgement

I wish to thank the brave students of my experimental class for motivating me to think of these questions, for sticking with me when the questions were impossible to understand, and for making many improvements to my original answers.

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

Webless Literate Programming

Jim Fox

Abstract

This article introduces c-~~web~~ (*no-web*, for short) as an alternative to the CWEB 'literate programming' system. c-~~web~~ is a method which allows a programmer to both `tex` (format) and `cc` (compile) the same source, without the need for preprocessors.

What is c-~~web~~

In **c** all comments begin with the characters '/*' and end with the characters '*/'. c-~~web~~ is a macro package that TeXs all comments, 'verbatims' all the code, and uses the comment delimiters to switch between the two modes. A c-~~web~~ program can be compiled directly by **c** and can be formatted directly by TeX. It has the advantage of high portability, while providing fully TeX'd comments, page headers and footers, and a table of contents.