Macros

FIFO and LIFO sing the BLUes*

Kees van der Laan

Abstract

FIFO, FirstInFirstOut, and LIFO, LastInFirstOut, are well known techniques for handling sequences. In TeX macro writing they are abundant but are not easily recognized as such. TeX templates for FIFO and LIFO are given and their use illustrated. The relation with Knuth's `\dolist`, answer Exercise 11.5, and `\ctest`, p. 376, is given.

Keywords: Education, FIFO, LIFO, list processing, macro writing, plain TeX.

Introduction

It started with the programming of the Tower of Hanoi in TeX, van der Laan (1992a). For printing each tower the general FIFO — First--In-First-Out[1] — approach was considered.[2] In literature (and courseware) the programming of these kind of things is done differently by each author, inhibiting intelligibility. In pursuit of Wirth (1976), TeX templates for the FIFO (and LIFO) paradigm will hopefully improve the situation.

In this article we will see various slightly different implementations of the basic FIFO principle.

FIFO

In the sequel, I will restrict the meaning of FIFO to an input stream which is processed argument-wise. FIFO can be programmed in TeX as template

```
\def\fifo #1%
   {\ifx\ofif#1\ofif\fi\process#1\fifo}
\def\ofif #1\fifo{\fi}
```

The `\fifo` command calls a macro `\process` that handles the individual arguments. Often you can copy `\fifo` straight out of this article, but you have to write a version of `\process` that is specific to your application.

To get the flavor.

Length of string. An alternative to Knuth's macro `\getlength`, *The TeXbook* p. 219, is obtained via the use of `\fifo` with

```
\newcount\length
\def\process #1{\advance\length \by1 }
```

Then `\fifo aap   noot\ofif \number\length` yields the length 7.[3]

Number of asterisks. An alternative to Knuth's `\atest`, *The TeXbook*, p. 375, for determining the number of asterisks, is obtained via `\fifo` with

```
\newcount\acnt
\def\process #1%
   {\if*#1\advance\acnt by1 \fi}
```

Then `\fifo abc*de*\ofif \number\acnt` yields the number of asterisks: 2.[4]

Vertical printing. David Salomon treats the problem of vertical printing in his courseware. Via an appropriate definition of `\process` and a suitable invocation of `\fifo` it is easily obtained.

```
\def\process #1{\hbox{#1}}
\vbox{\offinterlineskip\fifo abc\ofif}
```

yields a
b
c.

Tower of Hanoi. Printing of a tower can be done via

```
\def\process #1%
   {\hbox to3ex{%
    \hss\vrule width#1ex height1ex\hss}}
\vbox{\baselineskip1.1ex\fifo12\ofif}
```

Termination. For the termination of the tail recursion the same TeXnique as given in the *The TeXbook*, p. 379, in the macro `\deleterightmost`, is used. This is elaborated as `\break` in Fine (1992), in relation to termination of the loop. The idea is that when `\ofif` is encountered in the input stream, that is, when `\ifx\ofif#1...` is true, all tokens in the macro up to and including `\fifo` — the start for the next level of recursion — are gobbled by a

---

*: Earlier versions appeared in MAPS 92.1, proceedings EuroTeX '92, and *TUGboat* 14.1. BLU is Ben Lee User of the *The TeXbook* fame. It makes the title sing, I hope.

1: See Knuth (1968), section 2.2.1.

2: In the Tower of Hanoi article Knuth's list data-structure was finally used — TeXbook Appendix D.2 — with FIFO inherent.

3: Insert `\obeyspaces` when the spaces should be counted as well.

4: As the reader should realize, this works correctly when there are first level asterisks *only*. For counting at all levels automatically, a more general approach is needed, see Knuth's `\ctest`, p. 376.

subsequent call to \ofif.[5] Because the matching \fi is gobbled too, this token is inserted via the replacement text of \ofif. This TEXnique is better than Kabelschacht's, (1987), where the token preceding the \fi is expanded after the \fi via the use of \expandafter. When this is applied the exchange occurs at each level in the recursion. It also better than the \let\next=... TEXnique, which is used in the *The TEXbook*, for example in \iterate, p. 219, because there are no assignments.

My first version had the two tokens after \ifx reversed — a cow flew by — and made me realize the non-commutativity of the *first level* arguments of TEX's conditionals. For example, \ifx aa\empty... differs from \ifx\empty aa..., and \if\ab\aa... from \if\aa\ab..., with \def\aa {aa}, \def\ab{ab}. In math, and in programming languages like Pascal, the equality relation is commutative,[6] and no such thing as expansion comes in between. When not alert with respect to expansion, TEX's \if-s can surprise you.

The \fifo macro is a basic one. It allows one to proceed along a list — at least conceptually — and to apply a (user) specified process to each list element. By this approach the programming of going through a list is *separated* from the various processes to be applied to the elements.[7] It adheres to the *separation of concerns* principle, which I consider fundamental.

The input stream is processed argumentwise, with the consequence that first level braces will be gobbled. Beware! Furthermore, no outer control sequences are allowed, nor \par-s. The latter can be permitted via the use of \long\def.

A general approach — relieved from the restrictions on the input stream: *every token* is processed until \ofif — is given in the *The TEXbook* answer to Exercise 11.5 (\dolist...) and on p. 376 (\ctest...). After adaptation to the \fifo notation and to the use of macros instead of token variables, Knuth's \dolist comes down to

```
\def\fifo
   {\afterassignment\tap \let\nxt= }
\def\tap
   {\ifx\nxt\ofif\ofif\fi\process
    \nxt\fifo}
\def\ofif#1\fifo{\fi}
```

This general approach is indispensable for macro writers. My less general approach can do a lot already, for particular applications, as will be shown below. But, ... beware of its limitations.

Variations. The above \fifo can be seen as a template for encoding tail recursion in TEX, with arguments taken from the input stream one after another. An extension is to take two arguments from the input stream at a time, with the second argument to look ahead, via

```
\def\fifo #1#2%
   {\process#1\ifx\ofif#2
    \ofif\fi\fifo#2}
\def\ofif#1\ofif{\fi}
```

Note the systematics in the use of the parameter separator in \ofif; here \ofif and in the previous macro \fifo, the last token of the replacement text. Although the principle of looking ahead with recursion is abundant in computer programming, a small example to illustrate its use is borrowed from Salomon: delete last character of argument. It is related to \deleterightmost, *The TEXbook* p. 379. Effective is the following, where a second parameter for \fifo is introduced to look ahead, which is inserted back when starting the next recursion level.

```
\def\gobblelast #1{\fifo#1\ofif}
\def\fifo #1#2%
   {\ifx\ofif#2\ofif\fi#1\fifo#2}
\def\ofif#1\ofif{\fi}
```

Then \gobblelast{aap} will yield aa.
And what about recursion without parameters? A nice example of that is a variant implementation of Knuth's \iterate of the \loop, *The TEXbook*, p. 219

```
\def\iterate
   {\body%
    \else\etareti%
    \fi%
    \iterate}
\def\etareti #1\iterate{\fi}
```

This \iterate contains only 5 tokens in contrast with Knuth's 11. The efficiency and the needed memory is determined by the number of tokens in \body, and therefore this 5 vs. 11 is not relevant. The idea behind including this variant here is that the FIFO principle can lead to simple encoding of tail recursion even when no arguments are processed.

Variable number of parameters. TEX macros can take at most 9 parameters. The above \fifo macro can be seen as a macro which is relieved from that

---

5: In contrast with usual programming of recursion start with the infinite loop, and then insert the \if...\ofif\fi.

6: So are TEX's \if-s after expansion.

7: If a list has to be *created*, Knuth's list data-structure might be used, simplifying the execution of the list. See *The TEXbook* Appendix D.2.

restriction. Every group, or admissible token, in the input stream after \fifo up to and including \ofif, will become an argument to the macro. When the \ofif token is reached, the recursion — that is reading of arguments — will be terminated.[8]

Unknown number of arguments. Tutelaers (1992), as mentioned by Eijkhout (1991), faced the problem of inputting a chess position. The problem is characterized by an unspecified number of positions of pieces, with for the pawn positions the identification of the pawn generally omitted. Let us denote the pieces by the capital letters K(ing), Q(ueen), B(ishop), (k)N(ight), R(ook), and P(awn), with the latter symbol default. The position on the board is indicated by a letter a, b, c, ..., or h, followed by a number, 1, 2, ..., or 8. Then, for example,

```
\position{Ke1, Qd1, Na1, e2, e4}
```

should entail the invocations

```
\piece{K}{e1}\piece{Q}{d1}\piece{N}{a1}
\piece{P}{e2}\piece{P}{e4}
```

This can be done by an appropriate definition of \position, and an adaptation of the \fifo template, via

```
\def\position #1%
   {\fifo#1,\ofif,}
\def\fifo #1,%
   {\ifx\ofif#1\ofif\fi%
    \process#1\relax\fifo}
\def\ofif #1\fifo{\fi}
\def\process #1#2#3%
   {\ifx\relax#3%
    \piece{P}{#1#2}\else\piece#1{#2#3}\fi}
```

With the following definition (simplified in relation to Tutelaers)

```
\def\piece #1#2{ #1-#2}
```

we get K-e1 Q-d1 N-a1 P-e2 P-e4.

For an unknown number of arguments at two levels see the Nested FIFO section.

Citation lists. In a list of citations it is a good habit to typeset three or more consecutive numbers as a range. For example 1, 2, 3 as 1–3. This must be done via macros when the numbers are represented by symbolic names, which get their value on the fly. In general the sequence must be sorted[9] before typesetting. This has been elaborated by Arseneau (1992) in a few LaTeX styles, and for plain TeX by myself. I used the FIFO paradigm in the trivial, stepping-stone, variant of typesetting an explicit non-descending sequence in range notation. The resulting 'process' macro could be used in the general case, once I realized that FISO—Firts-In--Smallest-Out—was logically related to FIFO: the *required* elements are yielded one after the other, whether the first, the last, the smallest, or ...you name it. Perhaps this is a nice exercise for the reader. For a solution see van der Laan (1993).[10]

Vowels, voilà. Schwarz (1987) coined the problem to print vowels in bold face.[11] The problem can be split into two parts. First, the general part of going character by character through a string, and second, decide whether the character at hand is a vowel or not.

For the first part use \fifo (or Knuth's \dolist). For the second part, combine the vowels into a string, aeiou, and the problem can be reduced to the question ⟨*char*⟩ ∈ aeiou? Earlier, I used this approach in searching a card in a bridge hand, van der Laan (1990, the macro \strip). That was well-hidden under several piles of cards, I presume? The following encoding is related to \ismember, *The TeXbook*, p. 379

```
\newif\iffound
%% locate #1 in #2
\def\loc #1#2%
   {\def\locate ##1#1##2\end%
      {\ifx\empty##2%
         \empty\foundfalse
        \else\foundtrue\fi}%
    \locate#2#1\end}
\def\process #1%
   {\uppercase{\loc#1}{AEIOU}%
    \iffound{\bf#1}\else#1\fi}
```

Then \fifo Audacious\ofif yields **A**ud**a**c**iou**s.

Variation. If in the invocation \locate #2#1 a free symbol is inserted between #2 and #1,

8: Another way to circumvent the 9 parameters limitation is to associate names to the quantities to be used as arguments, let us say via def's, and to use these quantities via their names in the macro. This is Knuth's parameter mechanism and is functionally related to the so-called keyword parameter mechanism of command languages, and for example ADA.

9: The sorting of short sequences within TeX has been elaborated by Jeffrey (1990), and myself in Syntactic Sugar.

10: However, in my later BLUe's Bibliography, this is no longer necessary because of the one-pass job and the inherent simpler approach.

11: His solution mixes up the picking up of list elements and the process to be applied. Moreover, his nesting of \if-s in order to determine whether a character is a vowel or not, is not elegant. Fine (1992)'s solution, via a switch, is not elegant either.

then `\loc` can be used to locate substrings.[12] And because $\{string_1 \in string_2\} \wedge \{string_2 \in string_1\} \Rightarrow string_1 = string_2$, the variant can be used for the equality test for strings. See also the Multiple FIFO subsection, for general and more effective alternatives for equality tests of strings.

Processing lines. What about processing lines of text? In official, judicial, documents it is a habit to fill out lines of text with dots.[13] This can be solved by making the end-of-line character active, with the function to fill up the line. A general approach where we can `\process` the line, and not only append to it, can be based upon `\fifo`.

One can wonder, whether the purpose can't be better attained, while using TeX as formatter, by filling up the last line of paragraphs by dots, because TeX justifies with paragraphs as units.

In the *The TeXbook* the example about processing lines is writing answers of the exercises to the file `answers.tex`, line by line, p. 422. The given `\copytoblankline` can be recast in FIFO terms as

```
\def\copytoblankline
   {\begingroup\setupcopy\fifol}
{\obeylines\gdef\fifol#1
  {\ifx\empty#1\empty\lofif\fi
   \processl{#1}\fifol}}
\def\lofif #1\fifol{\fi\endgroup}
\def\processl #1%
   {\immediate\write\ans{#1}}
```

Processing words. What about handling a list of words? This can be achieved by modifying the `\fifo` template into a version which picks up words, `\fifow`, and to give `\processw` an appropriate function.

```
\def\fifow #1 %
   {\ifx\wofif#1\wofif\fi
    \processw{#1}\ \fifow}
\def\wofif #1\fifow{\fi}
```

Underlining words. In print it is uncommon to emphasize words by underlining. Generally another font is used, see discussion of Exercise 18.26 in the *The TeXbook*. However, now and then people ask for (poor man's) underlining of words. The following `\processw` definition underlines words picked up by `\fifow`. Then

```
\def\fifow #1 %
   {\ifx\wofif#1\wofif\fi
    \processw{#1}\ \fifow}
\def\wofif #1\fifow{\fi}
```

12: Think of finding 'bb' in 'ab' for example, which goes wrong without the extra symbol.

13: The problem was posed at EuroTeX '91 by Theo Jurriens.

```
\def\processw #1%
   {\vtop{\hbox{\strut#1}\hrule}}
%%
\fifow leentje leerde lotje lopen
langs de lange lindenlaan \wofif\unskip
```

yields <u>leentje</u> <u>leerde</u> <u>lotje</u> <u>lopen</u> <u>langs</u> <u>de</u> <u>lange</u> <u>lindenlaan</u>.

Nested FIFO

One can nest the FIFO paradigm. For processing lines word by word, or words character by character.

Words character by character. Exercise 11.5, can be solved by processing words character by character. A solution to a slightly simplified version of the exercise reads

```
\fifow Though  exercise \wofif \unskip.
```

with

```
\def\processw #1{\fifo#1\ofif}
\def\process #1{\boxit#1}
\def\boxit #1%
   {\setbox0=\hbox{#1}\hbox{\lower\dp0%
    \vbox{\offinterlineskip
          \hrule
          \hbox{\vrule\phantom#1\vrule}
          \hrule}}}
```

yields .

In the spirit of `\dolist...`, Exercise 11.5, is (variant neglecting word structure)

```
\def\fifo
   {\afterassignment\tap\let\nxt= }
\def\tap {\ifx\nxt\ofif\ofif
          \fi\process\nxt\fifo}
\def\ofif #1\fifo{\fi}
\def\process #1%
   {\if\space\nxt\
    \else\boxit#1\fi}
\fifo Though    exercise\ofif.
```

with the same result .

Mark up natural data. Data for `\h(v)align` needs & and `\cr` marks. We can get plain TeX to append a `\cr` at each input line, *The TeXbook* p. 249. An extension of this is to get plain TeX to insert `\cs`-s, column separators, and `\rs`-s, row separators, and eventually to add `\lr`, last row, at the end, in natural data. For example prior to an invocation of `\halign`, one wants to get plain TeX to do the transformation

P*ON
DEK* ⇒ P\cs*\csO\csN\rsD\csE\csK\cs*\lr

This can be done via

```
\vbox{\hbox{P*ON}\kern.5ex
      \hbox{DEK*}} \,\Rightarrow\,
\bdata P*ON
DEK*
```

```
\edata \markup\data
\vcenter{\hbox{\data}}
```

with

```
\let\ea=\expandafter
\def\bdata {\bgroup\obeylines\store}
\def\store #1\edata{\egroup\def\data{#1}}
\def\markup #1%
   {\ea\xdef\ea#1\ea{\ea\fifol#1\lofif}}
```

and auxiliaries

```
{\catcode`\^^M=13
\gdef\fifol #1^^M#2%
    {\fifo#1\ofif%
     \ifx\lofif#2\noexpand\lr\lofif
     \fi\noexpand\rs\fifol#2}}
\def\lofif #1\lofif{\fi}
\def\fifo #1#2{%
    #1\ifx\ofif#2\ofif
    \fi\noexpand\cs\fifo#2}
\def\ofif #1\ofif{\fi}
\def\cs{{\sevenrm{\tt\char92}cs}}
\def\rs{{\sevenrm{\tt\char92}rs}}
\def\lr{{\sevenrm{\tt\char92}lr}}
```

The above came to mind when typesetting crosswords,[14] van der Laan (1992b,d),[15] while striving after the possibility to allow natural input, independent of \halign processing.

Multiple FIFO

What about FIFO for more than one stream?[16] For example comparing strings, either for equality or with respect to lexicographic ordering? Eijkhout (1992, p. 137, 138) provided for these applications the macros \ifAllChars...\Are...\TheSame, and \ifallchars...\are...\bfore. The encodings are focused at mouth processing. The latter contains many \expandafter-s.

A basic approach is: loop through the strings character by character, and compare the characters until either the assumed condition is no longer true, or the end of either one of the strings, has been reached.

Equality of strings. The TEX-specific encoding, where use has been made of the property of \ifx for control sequences, reads

```
\def\eq #1#2%
   {\def\st{#1}\def\nd{#2}
    \ifx\st\nd\eqtrue\else\eqfalse\fi}
```

with auxiliary \newif\ifeq.

14: With *, or ␣, given an appropriate function.

15: In (1992d) I set the puzzles via direct use of nested FIFO. No \halign use nor mark up phase.

16: For simplicity the streams are stored in def-s, because \read inputs lines.

As a stepping stone for lexicographic comparison, consider the general encoding

```
\def\eq #1#2%
   {\continuetrue\eqtrue
    \loop
       \ifx#1\empty\continuefalse\fi
       \ifx#2\empty\continuefalse\fi
    \ifcontinue
       \nxte#1\nxtt \nxte#2\nxtu
       \ifx\nxtt\nxtu
       \else\eqfalse\continuefalse\fi
    \repeat
    \ifx\empty#1
       \ifx\empty#2\else\eqfalse\fi
    \else\eqfalse\fi}
```

with auxiliaries

```
\newif\ifcontinue \newif\ifeq
\def\nxte #1#2%
   {\def\pop ##1##2\pop%
      {\gdef #1{##2} \gdef#2{##1}}%
       \ea\pop#1\pop}
```

Then

```
\def\t{abc} \def\u{ab} \eq\t\u
\ifeq$abc=ab$\else$abc\not=ab$\fi
```

yields $abc \neq ab$.

Lexicographic comparison. Assume that we deal with lower case and upper case letters only. The encoding of \sle — String Less or Equal — follows the same flow as the equality test, \eq, but differs in the test, because of TEX's expansion mechanisms

```
% #1, #2 are def's
\def\sle #1#2%
  {\global\sletrue\global\eqtrue
   {\continuetrue
   \loop
      \ifx#1\empty\continuefalse\fi
      \ifx#2\empty\continuefalse\fi
   \ifcontinue
      \nxte#1\nxtt\nxte#2\nxtu
      \ea\ea\ea\lle\ea\nxtt\nxtu
   \repeat}
   \ifeq\ifx\empty#2\ifx\empty#1
        \else\global\slefalse\fi\fi
   \fi}
```

with auxiliaries (\lle=Letter Less or Equal)

```
\newif\ifcontinue
\global\newif\ifsle
\global\newif\ifeq
\def\nxte #1#2%
   {\def\pop##1##2\pop%
       {\xdef#1{##2}%
        \xdef#2{##1}}%
    \ea\pop#1\pop}
\def\lle #1#2%
   {\uppercase{\ifnum`#1=`#2}
    \else\continuefalse\global\eqfalse
    \uppercase{\ifnum`#1>`#2}{}
              \global\slefalse\fi
    \fi}
```

For example

```
\def\t{ABC} \def\u{ab} \sle\t\u
\ifsle$ABC\le ab$\else$ABC>ab$\fi
```

yields $ABC > ab$;

```
\def\t{aa} \def\u{a} \sle\t\u
\ifsle$aa\le a$\else$aa>a$\fi
```

yields $aa > a$;

```
\def\t{aa} \def\u{b} \sle\t\u
\ifsle$aa\le b$\else$aa>b$\fi
```

yields $aa \le b$;

```
\def\t{noo} \def\u{apen} \sle\t\u
\ifsle$noo<apen$\else$noo>apen$\fi
```

yields $noo > apen$.

The above can be elaborated with respect to \read for strings each on a separate file, to strings with accented letters, to the inclusion of an ordering table, and in general to sorting. Some of the mentioned items will be treated in Sorting in BLUe, to come.

LIFO

A modification of the \fifo macro — \process {#1} invoked at the end instead of at the beginning — will yield the LastInFirstOut template. Of course LIFO can be applied to reversion on the fly, without explicitly allocating auxiliary storage.[17]

```
\def\lifo #1#2\ofil%
   {\ifx\empty#2\empty\ofil\fi%
    \lifo#2\ofil\process#1}
\def\ofil #1\ofil{\fi}
```

The test for emptyness of the second argument is similar to the TeXnique used by Knuth in \displaytest, *The TeXbook* p. 376: \if!#3!....

With the identity — \def\process #1{#1}, or the invoke \process #1 replaced by #1[18] — the template can be used for reversion on the fly

For example \lifo aap\ofil yields paa.[19]

Change of radix. In the *The TeXbook* a LIFO exercise is provided at p. 219: print the digits of a number in radix 16 representation. The encoding is based upon the property

$$d_k = (N \div r^k) \bmod r, \quad k = 0, 1, \ldots, n,$$

with radix r, coefficients d_k, and the number representation

$$N = \sum_{k=0}^{n} d_k\, r^k.$$

There are two ways of generating the numbers d_k: starting with d_n, or the simpler one starting with d_0, with the disadvantage that the numbers are generated in reverse order with respect to printing. The latter approach is given in *The TeXbook* p. 219. Adaptation of the LIFO template does not provide a solution much different from Knuth's, because the numbers to be typeset are generated in the recursion and not available in the input stream.

Further reading

Zalmstra and Rogers (1989), apply the FIFO technique to a list of figures — or floating bodies — in order to merge the list appropriately with the main vertical list in the output routine. This is beyond the scope of this paper.

Acknowledgements

Włodek Bzyl and Nelson Beebe are kindly acknowledged for their help in clearing up the contents and correcting my use of English, respectively.

Conclusion

In looking for a fundamental approach to process elements sequentially — not to confuse with list processing where the list is also built up, see *The TeXbook* Appendix D.2, or with processing of *every* token in the input stream, see Exercise 11.5 or p. 376 — TeX templates for FIFO and LIFO emerged.

The templates can be used for processing lines, words or characters. Also processing of words line

17: Johannes Braams drew my attention to Knuth and MacKay (1987), which contained among others \reflect ... \tcelfer. They compare #1 with \empty, which is nice. The invocation needs an extra token, \empty — a so-called sentinel, see Wirth (1976) — to be included before \tcelfer, however. (Knuth and Mackay hide this by another macro which invokes \reflect... \empty\tcelfer). My approach requires at least one argument, with the consequence that the empty case must be treated separately, or a sentinel must be appended after all.

18: Remember the stack size limitations.

19: Note that Knuth's test \if!\#3!... goes wrong for #3 equals !, and similarly my use of the idea goes wrong for #2 equals \empty, which is not 'empty.' Given the context those situations don't occur, however.

by line, or characters word by word, can be handled via nested use of the FIFO principle.

The FIFO principle along with the look ahead mechanism is applied to molding natural data into representations required by subsequent TEX processing.

Courseware might benefit from the FIFO approach to unify answers of the exercises of the macro chapter.

TEX's `\ifx...` and `\if...` conditionals are non-commutative with respect to their *first level* operands, while the similar mathematical operations are, as are the operations in current high-level programming languages.

Multiple FIFO, by comparing strings lexicographically, has been touched upon.

References

[1] Arseneau D (1992): overcite.sty, drftcite.sty, cite.sty. From the file server.

[2] Eijkhout V (1991): TEX by Topic. Addison--Wesley.

[3] Fine, J (1992): Some basic control macros for TEX. *TUGboat* 13, no. 1, 75–83.

[4] Hendrickson A (priv. comm.)

[5] Jeffrey A (1990): Lists in TEX's mouth. *TUGboat* 11, no. 2, 237–244.

[6] Kabelschacht A (1987): `\expandafter` vs. `\let` and `\def` in conditionals and a generalization of plain's `\loop`. *TUGboat* 8, no. 2, 184–185.

[7] Knuth D.E (1968): The Art of Computer Programming. 1. Fundamental Algorithms. Addison-Wesley.

[8] Knuth D.E (1984): The TEXbook. Addison--Wesley.

[9] Knuth D.E, P Mackay (1987): Mixing righttoleft texts with lefttoright texts. *TUGboat* 7, no. 1, 14–25.

[10] Laan C.G van der (1990): Typesetting Bridge via TEX. *TUGboat* 11, no. 2, 91–94. Also MAPS 91.2, 51–62.

[11] Laan C.G van der (1992a): Tower of Hanoi, revisited. *TUGboat* 13, no. 1, 91–94. (Also MAPS 92.1, 125–127.)

[12] Laan C.G van der (1992b): Typesetting Crosswords via TEX. EuroTEX '92, 217–224. (Also MAPS 92.1, 128–131.)

[13] Laan C.G van der (1992c): Table Diversions. EuroTEX '92, 191–211. (Also a little adapted in MAPS 92.2, 115–128. Macros on CTAN)

[14] Laan C.G van der (1992d): Typesetting Crosswords via TEX, revisited. MAPS 92.2, 145–146. (Macros on CTAN.)

[15] Laan C.G van der (1992e): Syntactic Sugar. MAPS92.2, 130-136. (Also TUG '93, *TUGboat* 14, no. 3, 310 – 318. Abridged GUST bulletin 1.)

[16] Laan, C.G van der (1993): Typesetting number sequences. MAPS 93.1, 145 – 148.

[17] Laan, C.G van der (1993): Sorting in BLUe. MAPS 93.1, 149 – 170. (Abridged TUG '93, *TUGboat* 14, no. 3, 319 – 328. For heap sort coding in plain TEX, see MAPS92.2, 137–138. Macros on CTAN.)

[18] Salomon D (1992): Advanced TEX course: Insights & Hindsights, MAPS 92 Special. ≈500p.

[19] Schwarz N (1987): Einführung in TEX. Addison-Wesley.

[20] Tutelaers P (1992): A font and a style for typesetting chess using LATEX or TEX. *TUGboat* 13, no. 1, 85–90.

[21] Wirth N (1976): Algorithms + Data Structures = Programs. Prentice-Hall.

[22] Zalmstra J, D.F Rogers (1989): A page make-up macro. *TUGboat* 10, no. 1, 73–81.