Sorting in TEX's Mouth[*]

Bernd Raichle

Stettener Str. 73
D-73732 Esslingen, FRG
`raichle@Informatik.Uni-Stuttgart.de`

Abstract

TEX's macro processor, the so-called *mouth,* can be used to perform very complex tasks. Because this part of TEX's programming language is as powerful as a Turing machine, it is possible to implement algorithms using only TEX's mouth.
I will show how sorting algorithms can be implemented in a straight-forward and very elegant and understandable way using only macros and macro expansion TEXniques.

1 Motivation

While reading Jeffrey's paper [2] about list processing in TEX's mouth in 1990, I figured I understood his macros and the underlying ideas — at the beginning, at least. My first attempts to implement a quicksort algorithm based on his insertion sort macros failed disastrously for a simple reason: I had a wrong model for TEX's macro processor. I thought in terms of a procedural or functional model in which a macro is seen as a *function* reading some tokens as its arguments and returning a token list as its function value, instead of thinking of it as a simple *replacement* of tokens by other tokens.

In the summer of 1993 with more experience in TEX macro programming, I retried the implementation of a quicksort algorithm with more success. Coincidentally at this time van der Laan's papers [6, 7] appeared, the first including an attempt at mouth-only processing, the second with his version of multi-purpose sorting macros.

This paper is my answer to van der Laan's questions about the usefulness and the need for mouth processing.

2 TEX's programming language

Users of TEX have different models for TEX. Depending on the user's needs, different parts of TEX and different abstraction levels of functionality are useful and appropriate. Someone who uses TEX as a text formatter via LATEX needs a different understanding than someone who uses TEX as a programming language writing macros for complicated tasks.

For a user, focussing on the programming language TEX, the language can be divided in two major parts. On one side, TEX contains the *mouth,* a macro processor providing a macro language with its own characteristics. On the other side, we have the *stomach* of TEX, the language part, where all commands with side effects, such as all assignments and text output or `dvi` output commands, are executed. The programming language realized by the stomach is incomplete because it misses control structures such as conditionals or loops.

The usable programming language has to consist of both parts because the assignment capabilities of the stomach are needed in order to define macro definitions and to read or write text, i.e., to produce an output. The mouth is necessary for all tasks that needs an iterative application or a recombination of input tokens. TEX's stomach uses the macro processor for almost all commands to scan the command arguments. Additionally, while scanning the arguments of many stomach commands, such as `\write`, `\edef`, count or dimension register assignments, all tokens are expanded. Thus stomach operations are not allowed in these places, leading to the problem of *fragile* commands in *moving* arguments. This is partly taken care of with LATEX's `\protect`.

Because of the importance of TEX's mouth, the rest of this paper focusses on the macro processor part in more detail.

2.1 TEX's macro processor

TEX's mouth realizes a macro language operating on token lists.[1] A token references either a primitive command or a macro definition and is built from the characters of the input files in TEX's *eyes* and *mouth* using a fixed set of rules [5, pp. 37ff].

The mouth reads one token after another from the currently active input, a file or a token list, and tries to expand each token. If the token read is unexpandable, i.e., it references a primitive with side effects, the expansion process is stopped

[*]Reprint from the Proceedings of the Eigth European TEX Conference, Gdańsk, Poland, September 26–30, 1994.
[1]Be aware that this statement and the use of the word *mouth* in this paper is imprecise as a result of Knuth's 'technique of deliberate lying'. More precisely, TEX's *gullet* is the macro processor, whereas the *mouth* is 'the process by which input files are converted to lists of tokens' [5, p. 267].

and the token is given to the stomach for further processing. If the token is expandable, i.e., it references a macro or a primitive without side effects, the mouth reads more tokens *without* expansion, if arguments are needed, and replaces the token read in the input by a list of tokens. This means that the next token is the first token of the replacement text because the original token is removed.

Comparing macros and macro expansion with procedures or functions in a procedural programming language, it is important to note that tokens building the arguments of a macro are *not* expanded at the time the macro is expanded.

2.2 Basic Operations

TEX's macro language only knows about a very restrictive set of basic operations on (token) lists implementable by very simple macro definitions. These basic operations[2] are

- get first element of a list
  `\def\Car#1#2\EOL{#1}`,
- get the rest of a list
  `\def\Cdr#1#2\EOL{#2\EOL}`,
- add a new element in front of a list
  `\def\Cons#1#2\EOL{{#1}#2\EOL}`,
- add a new element to the end of a list
  `\def\tCons#1#2\EOL{#2{#1}\EOL}`,
- append two lists
  `\def\Append#1\EOL#2\EOL{#1#2\EOL}`,
- and similar operations to get the n-th element of a list, add n elements, or append n lists to a list with $n < 9$.

Complex operations, such as

- get the length of a list,
- search for an element in a list,
- return the n-th element of a list,
- delete all elements equal to a specified element,
- reverse a list,
- apply an operation to all elements of a list,

and a lot more operations are not supported by simple macros, but have to be defined using more or less complex macro definitions. When looking on the list of complex operations, it should be obvious that they are based on an important concept: *iteration!*

2.3 Loops and Iteration

The macro language does not contain any loop primitives or any other primitives allowing iteration. The reason is simple: it is not necessary. In a macro language missing an iteration primitive, a loop can be easily implemented by using the macro token realizing the loop in its own replacement text. That is, iteration is implemented by using recursion.

If the replacement text of a macro contains a token referencing this macro, the expansion process in the mouth will expand the macro and expand the macro and expand the macro . . . .

[2] The lists in the examples are token sequences delimited by the token `\EOL`.

Example 1 We want to apply a macro `\Func` to all tokens of a list. A first definition for the iteration macro is

```
\def\Mapc#1{\Func{#1}\Mapc}
\def\Func#1{ (#1) }
```

Applying this macro to the token list `1234`, will yield the expansion sequence

```
\Mapc                   1 2 3 4
\Func{1}\Mapc             2 3 4
   (1)  \Mapc             2 3 4
   (1)  \Func{2}\Mapc       3 4
   (1)     (2)  \Mapc       3 4
   (1)     (2)  \Func{3}\Mapc 4
   (1)     (2)     (3)  \Mapc 4
  . . .
```

This first macro definition seems to solve our problem. But what will the macro `\Mapc` do after reading the last element 4? It will continue and will finally stop with an error message since we have implemented an endless loop.

Before solving this problem, a small change to the macro definition will show another iteration macro technique which can be used to collect temporary values.

Example 2 We want to collect the application of a macro `\Func` to each element in a list without applying this macro, until we have iterated over the complete list.

```
\def\MapCar{\DoMapCar{}}
\def\DoMapCar#1#2{\DoMapCar{#1\Func{#2}}}
```

Before the iteration starts, the value of this argument, in which we collect the result, is initialised with the empty list. Applying the new macro to the list `1234` yields the sequence

```
\MapCar                         1 2 3 4
\DoMapCar{}                     1 2 3 4
\DoMapCar{\Func{1}}               2 3 4
\DoMapCar{\Func{1}\Func{2}}         3 4
\DoMapCar{\Func{1}\Func{2}\Func{3}} 4
   . . .
```

An advantage of `\MapCar` in comparison to `\Mapc` is important: because the collected result is an argument of `\DoMapCar`, it is possible to enhance this macro by applying additional operations after the iteration is completed.

To make the two macros perfect for common use, it is necessary to add tests checking for the end of the list which terminates the recursion.

Example 3 The definitions of the two macros `\Mapc` and `\MapCar` are completed by adding `\if...` comparisons checking for the end of the argument token list. I will use the token `\relax` in all following macro examples as the end of list marker. Be aware of this in case you want to use these macros with lists containing `\relax` as a normal element.

```
\def\Mapc#1{\DoMapc#1\relax}
\def\DoMapc#1{%
  \ifx\relax#1%  end of list?
  \else
    \ReturnFi{\Func{#1}\DoMapc}%
  \fi}

\def\MapCar#1{\DoMapCar{}#1\relax}
\def\DoMapCar#1#2{%
  \ifx\relax#2%  end of list?
    \ReturnElseFi{#1}%
  \else
    \ReturnFi{\DoMapCar{#1\Func{#2}}}
  \fi}

\def\ReturnFi#1\fi{\fi #1}
\def\ReturnElseFi#1\else#2\fi{\fi #1}

\def\Func#1{(#1)}% ... as an example
```

When adding `\if...` comparisons to complete these two macros, it is necessary to ignore all tokens of the false branch of the test including the `\else` and `\fi` tokens. Otherwise, the following iterations will use these tokens instead of the next element of the list as its arguments. Additionally, the definition of a macro without ignoring these tokens applied to a list will overflow some of TEX's internal stacks.

To avoid this problem, I have used the special macros `\Return...` in the `\Mapc` and `\MapCar` macros. In most cases `\expandafter` can be used instead of the `\Return...` macros to skip over the `\else` and `\fi` [3], but sometimes a 'slightly ridiculous sequence' of `\expandafters` is needed [1].

3 Sorting

Sorting is a basic tool whose algorithms and algorithm implementations are complex and interesting enough to use it for studying the advantages and disadvantages of a programming language. In the rest of this paper, I will explain the implementation of sorting algorithms in TEX's macro processor by using the technique shown in the `\MapCar` macro.

From the set of well-known internal sorting algorithms [4, pp. 73ff], such as sorting by insertion (straight insertion sort, Shell's sort), by selection (straight selection sort, Heapsort), by exchanging (bubble sort, shaker sort, Quicksort), and by merging (merge sort), I will describe and implement Quicksort in detail because this sorting method is appropriate and fast enough for larger lists.

3.1 Quicksort

The principle of the quicksort method is easily explained: Given is a field L with n elements. Choose a key value K. Partition the field L in two subfields L_l and L_r such that $\forall x \in L_l : x \leq K$ and $\forall x \in L_r : x \geq K$. Apply these partitioning steps to the subfields recursively until all subfields contain at most one element. The concatenation of all subfields is the sorted field.

A difficulty in implementing the quicksort method is the selection of the key value K at the beginning of each partitioning step. If K can be chosen in such a way that the two resulting subfields contain an equal number of elements, the execution time of the algorithm will be of order $n \log n$. In the worst case, one of the two subfields contains only one element, so that the execution time will be of order n^2. Because the search for the best value K in each subfield needs additional execution time, simpler selection methods are usually implemented: choose a random integer between the value of the first and the last element or consider a small sample and choose the median of this sample.

When implementing the quicksort method using arrays, the partitioning needs to be executed 'in place' because of memory restrictions. In those implementations elements are exchanged in pairs until the field is partitioned into the two subfields [4, pp. 114ff].

3.2 Quicksort and Lists

When trying to implement the quicksort method using TEX's macro processor, we have to use the description above for (token) lists. Ignoring the problem of selecting a good key value K, we can use the following algorithm:

Given is a list with more than one element. Choose the first element as the key value K; the two sublists L_l and L_r are empty. Compare each element x in the rest of the list with the key value K. If $x \leq K$, append it to L_l, otherwise to L_r.

Example 4 Given is a list $L = \{3, 6, 1, 8, 7, 2, 5, 0, 4\}$. We choose the first element (3) as key value and initialise the two sublists with $L_l = \{\}$ and $L_r = \{\}$. After iterating over all elements remaining in the list, the two sublists are $L_l = \{1, 2, 0\}$ and $L_r = \{6, 8, 7, 5, 4\}$.

If this partitioning step is applied recursively to all sublists, and at the end these sublists and key values are concatenated in the correct order, we get a sorted list.

An important fact is that a simple iteration is the only complex list operation in this algorithm, otherwise only primitive list operations (get first element, get the rest of a list, append lists) are used.

3.3 Quicksort Implementation

Using the technique of the `\MapCar` macro, an implementation of this algorithm is easy:

```
\def\QuickSort#1{\StartPartition#1\relax}
\def\StartPartition#1{%
  \ifx\relax#1%    % empty list?
  \else \ReturnFi{\DoPartition{#1}{}{}}%
  \fi}

\def\DoPartition#1#2#3#4{%
  \ifx\relax#4%    % end of rest?
    \ReturnElseifFiFi{\QuickSort{#2}%
                       {#1}%
                     \QuickSort{#3}}%
  \else\ifnum#4<#1 % element < key value?
    \ReturnElseFiFi
         {\DoPartition{#1}{#2{#4}}{#3}}%
  \else \ReturnFiFi
         {\DoPartition{#1}{#2}{#3{#4}}}%
  \fi\fi}
```

```
\def\ReturnFi#1\fi      {\fi #1}
\def\ReturnElseifFiFi#1\else#2\fi\fi
                        {\fi #1}
\def\ReturnFiFi#1\fi\fi {\fi\fi #1}
\def\ReturnElseFiFi#1\else#2\fi\fi
                        {\fi\fi #1}
```

When the macro `\QuickSort` is applied to the list `{{3}{6}{1}{8}{7}{2}{5}{0}{4}}` of Example 4 and `\tracingmacros` is set to a non-zero positive number, the following sequence can be observed in the lines of the `log` file where the `\QuickSort` macro is called:

```
{3}{6}{1}{8}{7}{2}{5}{0}{4}
{1}{2}{0}
{0}
{2}
{6}{8}{7}{5}{4}
{5}{4}
{4}
{8}{7}
{7}
```

This sequence is the result of the partitioning tree shown in Figure 1. Looking at the leaves of this tree, you will see the sorted list.

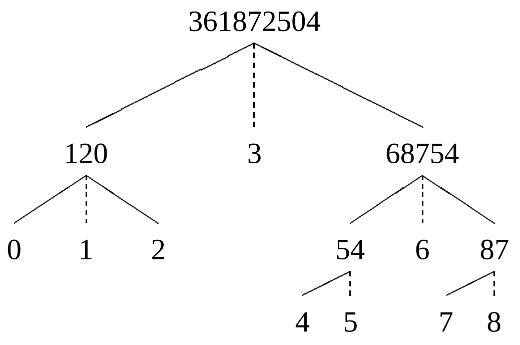

Figure 1: *Sublist partitions (solid lines) and the used key values K (dotted lines) for the example* `\QuickSort{361872504}`.

3.4 Necessary Improvements

Because the first implementation of quicksort uses a very simple scheme to choose the key values K, it is normal that tests will show worst cases of the execution time when these macros are applied to already sorted lists or to lists where all elements are equal.

40 elements	(a)	(b)	(c)	(d)
time	18	41	stack overflow	36

Table 1: *Execution time (in seconds) of the simple Quicksort implementation for a list with 40 elements.*

I have applied the first quicksort implementation to a list with 40 elements (a) in random order, (b) sorted in ascending order, (c) sorted in descending order, and (d) with equal elements. Table 1 shows the results obtained on a slow personal computer (Atari ST, 68 000/8 MHz).

Case (c), the sorted list in reverse order, is the worst case for this implementation: if the list contains more than some 30 elements, TEX will abort with an overflow of the parameter stack.

To overcome this problem, it is necessary to either use a better selection scheme for the key values, which is impossible without using more complex operations, or to ensure that the list and sublists are not sorted for *all* partitioning steps.

Example 5 If each odd numbered element of the list $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$ is appended to the tail and each even numbered element to the front of the list $B_0 = \{\}$, the result is the sequence of lists $B_1 = \{0\}$, $B_2 = \{1, 0\}$, $B_3 = \{1, 0, 2\}, \ldots, B_9 = \{7, 5, 3, 1, 0, 2, 4, 6, 8\}$.

A very simple method to disturb the order in a list is sketched in Example 5. Applying this method to the implementation when appending an element to one of the two sublists, a better behaviour for the worst cases is attained.

```
\def\QuickSort#1{\StartPartition#1\relax}
\def\StartPartition#1{%
  \ifx\relax#1%    % empty list?
  \else \ReturnFi{\DoPartitionI{#1}{}{}}%
  \fi}
```

```
\def\DoPartitionI#1#2#3#4{%
  \ifx\relax#4%    % end of rest?
    \ReturnElseifFiFi{\QuickSort{#2}%
                       {#1}%
                   \QuickSort{#3}}%
  \else\ifnum#4<#1 % element < key value?
    \ReturnElseFiFi
         {\DoPartitionII{#1}{#2{#4}}{#3}}%
  \else \ReturnFiFi
         {\DoPartitionII{#1}{#2}{#3{#4}}}%
  \fi\fi}
```

```
\def\DoPartitionII#1#2#3#4{%
  \ifx\relax#4%    % end of rest?
    \ReturnElseifFiFi{\QuickSort{#2}%
                       {#1}%
                   \QuickSort{#3}}%
  \else\ifnum#4>#1 % element > key value?
    \ReturnElseFiFi
         {\DoPartitionI{#1}{#2}{{#4}#3}}%
  \else \ReturnFiFi
         {\DoPartitionI{#1}{{#4}#2}{#3}}%
  \fi\fi}
```

A drawback of this disordering trick should be noted: whereas the first quicksort implementation is *stable*, i.e., elements with equal keys retain their original relative order, the new one is unstable.

3.5 Comparison with other Quicksort Implementations

In order to compare this fully expandable version of quicksort with other quicksort versions in TEX, I have used two other, not fully expandable implementations: line (I) in Table 2 represents the execution times for the shown `\QuickSort` macro, line (III) shows the result of Kees van der Laan's multi-purpose implementation [7]. Finally, I tried to implement an optimized quicksort version according to Knuth's description [4, S. 114ff], using a median of three values for the key values K in each partitioning step. Line (IV) shows the result for this version.

# elements	1	20	40	80	160	320
(I)	11	14	20	46	90	289
(II)	11	13	18	36	66	196
(III)	14	17	26	34	68	162
(IV)	10	13	16	25	45	93

Table 2: *Execution time (in seconds) of the four Quicksort implementations for disordered lists.*

Whereas the execution times of the two quicksort implementations (III) and (IV) are of order $n \log n$, the execution time of the 'mouth only' implementation (I) is of order n^2. The implementation uses the quicksort method — why is the execution time not proportional to $n \log n$? The answer to this question is surprising: In the partitioning loop, TEX has to scan the tokens of the two collected sublists as arguments of the `\DoPartition` macro and it has to skip these tokens in the false branches of the used conditions using the `\Return...` macros. Nonetheless the table shows that the presented `\Quicksort` macro is fast enough for short lists with less than 50–60 elements.

Using this analysis of the problem, it is possible to speed up the `\QuickSort` macros: We move the arguments for the recursive calls in `\DoPartitionI/II` to the end and replace the calls by new macros with the same argument structure. These new macros expand to the former macro sequences in the conditional branches with the necessary argument recombination. With these changes, TEX has to scan the sublists in `#2` and `#3` only twice instead of three times. The macros of the final version of `\QuickSort` are shown in the next section; line (II) of Table 2 contains the measured times.

3.6 Implementation Enhancements

The macros shown implement a quicksort algorithm suitable only for integers. After substituting the integer comparison tests `\ifnum#4<#1` (and `\ifnum#4>#1`) by a `\Compare` macro and using an appropriate definition for this macro, these macros can be used to sort other data types.

```
\def\QuickSort#1{\StartPartition#1\relax}
\def\StartPartition#1{%
  \ifx\relax#1%    % empty list?
  \else \ReturnFi{\DoPartitionI{#1}{}{}}%
  \fi}
```

```
\def\DoPartitionI#1#2#3#4{%
  \ifx\relax#4%    % end of rest?
    \ReturnElseifFiFi\DoQuickSort
  \else\Compare{#1}{#4}%
    \ReturnElseFiFi\PartitionGreaterII
  \else
    \ReturnFiFi\PartitionLessII
  \fi\fi
  {#1}{#2}{#3}{#4}}
```

```
\def\DoPartitionII#1#2#3#4{%
  \ifx\relax#4%    % end of rest?
    \ReturnElseifFiFi\DoQuickSort
  \else\Compare{#4}{#1}%
    \ReturnElseFiFi\PartitionLessI
  \else
    \ReturnFiFi\PartitionGreaterI
  \fi\fi
  {#1}{#2}{#3}{#4}}
```

```
\def\DoQuickSort#1#2#3#4{%
  \QuickSort{#2}\Func{#1}\QuickSort{#3}}
\def\PartitionLessI    #1#2#3#4{%
  \DoPartitionI{#1}{#2}{{#4}#3}}
\def\PartitionLessII   #1#2#3#4{%
  \DoPartitionII{#1}{#2}{#3{#4}}}
\def\PartitionGreaterI #1#2#3#4{%
  \DoPartitionI{#1}{{#4}#2}{#3}}
\def\PartitionGreaterII#1#2#3#4{%
  \DoPartitionII{#1}{#2{#4}}{#3}}
```

```
\def\Func#1{{#1}}
\def\Compare#1#2{\ifnum #1>#2\space}
```

The `\QuickSort` macros do not depend on the macro `\Compare` being fully expandable. However, only if `\Compare` is fully expandable will the resulting `\QuickSort` be fully expandable.

In the following example for a `\Compare` macro, a list of text words is sorted by the length of the typeset text using a roman font.

```
\def\Compare#1#2{\begingroup \rm
  \setbox0=\hbox{#1}%
  \setbox2=\hbox{#2}%
  \expandafter\endgroup\ifdim\wd0>\wd2 }
\def\Func#1{#1\par}
```

In order to allow the postprocessing of the sorted list, each element is handed to the macro `\Func`, which can be changed appropriately.

4 Conclusions

Is it necessary to restrict the implementation of the sorting algorithm to mouth processing? Or using van der Laan's words [6, p. 315]: 'Why don't [the authors] make clear the *need* for mouth processing, or should I say mouth optimizing?'

Generally, it is always necessary to *use* mouth processing because the mouth is the only language part of TEX processing macros with conditionals, and hence the necessary capabilities to provide iterations.

Additionally, it is necessary to *restrict* macro definitions to mouth processing, if the context of the macro usage forces it because TEX's stomach processes the macro in *expansion only mode*, e.g. in the argument of a `\write` or `\edef` command. There are even more examples of contexts in which only mouth processing is allowed: 1) Integers, dimensions or glue specifications can be built using macros specifying different parts. If one of these macros expands to an unexpandable token, e.g. `\relax` or another stomach command, TEX stops the scanning process and probably complains about a missing number. 2) `german.sty` [8] uses an active double quote character to allow the shortcut input `"a` for an `ä`. Because a double quote is also used in TEX to input numbers in hexadecimal notation, it is necessary to use mouth-only processing to decide whether the double quote introduces a hexadecimal number or not. 3) Inside the mouth command `\csname`... `\endcsname` all used control sequences have to be expandable.

At first glance, the restriction to mouth processing seems to be necessary only for expert TEX users writing some high-level macros using low-level macros and primitives. Thinking of a LATEX beginner who needs to insert `\protect` in some obscure places, it seems that the complex interactions between TEX's eyes, mouth, and stomach cannot be hidden from ordinary users.

Finally, in order to use the full power of the programming language TEX, it is necessary to get a better understanding of the capabilities and restrictions of its parts. Furthermore, since TEX's stomach will never be used without its mouth, it is best to start playing with TEX's macro processor.

5 Acknowledgements

I am pleased to thank Alan Jeffrey for his helpful comments on a preliminary version of this paper. In particular, his idea of introducing additional macros based on my analysis of the `\QuickSort` version (I) has led to the improved version (II). Thanks to Frank Tränkle and Jörg Heitkötter (–joke) for proofreading.

References

[1] Victor Eijkhout, Oral TEX: Erratum, *TUGboat*, **13**(1):75, 1992.

[2] Alan Jeffrey, Lists in TEX's Mouth, *TUGboat*, **11**(2):237–245, 1990.

[3] Alois Kabelschacht, `\expandafter` vs. `\let` and `\def` in Conditionals and a Generalization of plain's `\loop`, *TUGboat*, **8**(2):184–185, 1987.

[4] Donald E. Knuth, *The Art of Computer Programming*, Volume 3: Sorting and Searching, Addison-Wesley, Reading, Mass., 1973.

[5] Donald E. Knuth, *The TEXbook*, Addison-Wesley, Reading, Mass., 1986.

[6] Kees van der Laan, Syntactic Sugar, *TUGboat*, **14**(3):310–318, 1993.

[7] Kees van der Laan, Sorting within TEX, *TUGboat*, **14**(3):319–328, 1993.

[8] Hubert Partl, German TEX, *TUGboat*, **9**(1):70–72, 1988.