

E-TeX: Guidelines for Future TeX Extensions — revisited

Frank Mittelbach

Contents

1	Introduction	47
2	A short history of “Extended”-TeX engines	48
2.1	pTeX	48
2.2	ML-TeX	48
2.3	$\mathcal{N}\mathcal{T}\mathcal{S}/\varepsilon\mathcal{X}$ TeX	49
2.4	ε -TeX	49
2.5	Omega/Aleph	49
2.6	pdfTeX	49
2.7	X _Y TeX	49
2.8	LuaTeX	50
2.9	iTeX	50
3	Review of the issues raised in 1990	50
3.1	Line breaking	50
3.1.1	Line-breaking parameters	51
3.2	Spacing	51
3.3	Page breaking	52
3.4	Page layout	54
3.5	Penalties — measurement for decisions	55
3.6	Hyphenation	55
3.7	Box rotation	56
3.8	Fonts	56
3.9	Tables	57
3.10	Math	57
3.11	TeX’s language	58
4	Overcoming the mouth/stomach separation	59
4.1	A standard TeX solution	60
4.2	A LuaTeX solution	61
5	Conclusions	61

List of Figures

1	TeX-like engines evolution	48
2	Areas of concern in original article	50
3	Interword spacing	52
4	TeX’s box/glue/penalty model	53
5	Baseline to baseline spacing	55
6	The expl3 logo	58

Abstract

Shortly after Don Knuth announced TeX 3.0 I gave a paper analyzing TeX’s abilities as a typesetting engine. The abstract back then said:

Now it is time, after ten years’ experience, to step back and consider whether or not TeX 3.0

is an adequate answer to the typesetting requirements of the nineties.

Output produced by TeX has higher standards than output generated automatically by most other typesetting systems. Therefore, in this paper we will focus on the quality standards set by typographers for hand-typeset documents and ask to what extent they are achieved by TeX. Limitations of TeX’s algorithms are analyzed; and missing features as well as new concepts are outlined.

Now — two decades later — it is time to take another look and see what has been achieved since then, and perhaps more importantly, what can be achieved now with computer power having multiplied by a huge factor and, last but not least, by the arrival of a number of successors to TeX that have lifted some of the limitations identified back then.

1 Introduction

When I was asked by the organizers of the TUG 2012 conference to give a talk, I asked myself

What am I currently working on that could be of interest?

The answer I gave myself was: I’m working on ideas to resolve or at least lessen the issues around complex page layout; in particular mechanisms to re-break textual material in different ways so that you can, for example, evaluate different float placements in conjunction with different caption formats, or to float galley text in different ways around floats protruding into the galley.

All that goes way back in time: the issues were formulated more than 20 years ago in a paper I gave in 1990 in Texas: “E-TeX: Guidelines for future TeX extensions” [26]. Back then there were no answers to the issues raised. However, that was a long time ago; computers got faster and people invented various TeX extensions since then — and once in a while there are even new ideas.

So when I reread my paper from that time I thought that it would be a good idea to analyze the issues listed from the 1990 paper again and see what has been achieved since then.

This paper starts with a short overview of the history of TeX’s successor engines and the capabilities they added or improved. We will then re-analyze issues discussed two decades ago and evaluate their status.

We conclude with a summary of the findings and a suggestion for a way forward.

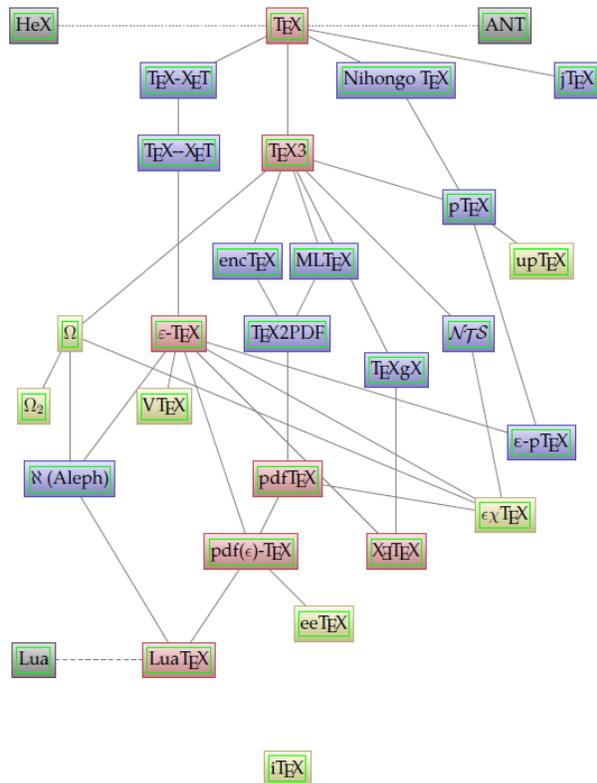


Figure 1: TeX-like engines evolution

2 A short history of “Extended”-TeX engines

Professor Donald Knuth developed the first version of the TeX program in 1978–79 [16, 17]. (The first specifications in writing date back to 1977; see [21].) Over the course of the next two years he improved and changed the program further and it then became known as TeX 82. This was the first widespread version of TeX, with documented source code [19] and a published manual [18], and people all over the world started using it.

Back then TeX used 7-bit fonts and typesetting in languages that required diacritics (as most European languages do) was difficult because, for example, hyphenation didn’t work properly in that case. Also, mixing of several languages in one document was impossible, at least if one wanted them to be hyphenated automatically.

Therefore, in 1989 a delegation of TeX users from Europe came to the Stanford meeting and presented Don with a proposal to extend TeX in several ways [31]. After several meetings and public discussions, Don recognized that he did not originally foresee a need for 8-bit input, and he agreed to extend TeX slightly to account for the needs of

the extended user base [20]. However, he only accepted those proposals that could be achieved with minimal adjustments to the existing program. If you compare *The TeXbook* before and after, you will have difficulty spotting the differences, because apart from the introduction of language support, nothing much changed. For example, my request for `\holdinginserts` was added because that was trivial to implement, but the suggestion for providing `\reconsiderparagraph` (to allow undoing the paragraph breaking to re-typeset it under different conditions) was rejected, as it would have meant more drastic updates to the paragraph-breaking algorithm.

This new version of TeX was called TeX 3.0, and shortly afterwards Don publicly announced that there would be no further version of TeX (except for bug fixes) and that his involvement in any future development of typesetting engines has ended with that version [20]. That announcement prompted me to analyze TeX’s abilities compared to high quality hand-typeset documents resulting in the paper given at the conference in Texas.

While TeX was thus officially frozen with version 3.0, other people started to build TeX engine variants to resolve one or another issue. The most influential ones are briefly outlined below; a nice overview of the more complete picture is given in [40], from which Figure 1 was taken with kind permission.

In the following we only discuss the major developments that contributed in one way or the other to an enriched feature set of the engine or have been influential in other ways.

2.1 pTeX

Typesetting in Japanese requires support of a huge character set and the ability to handle different typesetting directions. Thus early on developers in Japan created an extension to TeX that supports both Kanji (two-byte fonts) and proper vertical typesetting, in addition to TeX’s horizontally oriented approach. Early versions of pTeX predate TeX 3.0 [11, 32].

2.2 ML-TeX

One of the earliest attempts to modify TeX itself to handle the problem of multilingual typesetting was Michael Ferguson’s work on ML-TeX. Amongst other things it added a `\charsubdef` primitive that provided substitutions for accented characters. This way TeX’s hyphenation algorithm would be able to correctly hyphenate words with diacritics, even if the fonts used did not contain the characters as individual glyphs.

With the availability of T1-encoded fonts (containing most of the accented characters used in

“Western” languages as individual glyphs) ML- \TeX was no longer necessary for these languages. Nevertheless, it is still available in most engines, but needs to be explicitly enabled on the command line.

2.3 $\mathcal{N}\mathcal{T}\mathcal{S}/\varepsilon\mathcal{X}\mathcal{T}\mathcal{E}\mathcal{X}$

The $\mathcal{N}\mathcal{T}\mathcal{S}$ project (New Typesetting System) was inaugurated by DANTE (the German \TeX Users Group) in 1992. Its objective was to re-implement \TeX in a 100% compatible way in Java. While \TeX was frozen, $\mathcal{N}\mathcal{T}\mathcal{S}$ was to remain flexible and extensible. The project completed successfully in 2000, passing the trip test, and thus proving that a reimplementation of \TeX in a different language was possible. As it turned out though, full compatibility with \TeX resulted in code that was less modular than initially hoped for, so that adding any extensions or providing modifications of algorithms turned out to be far more difficult than initially anticipated. For this and a number of other reasons, $\mathcal{N}\mathcal{T}\mathcal{S}$ itself wasn’t developed any further.

$\varepsilon\mathcal{X}\mathcal{T}\mathcal{E}\mathcal{X}$ is a spin-off started around 2003 with the intention of developing a new Java-based system incorporating the experiences from $\mathcal{N}\mathcal{T}\mathcal{S}$, $\varepsilon\mathcal{T}\mathcal{E}\mathcal{X}$, pdf $\mathcal{T}\mathcal{E}\mathcal{X}$ and Omega. The project is represented on the web [1], but as of today it hasn’t left alpha stage.

2.4 $\varepsilon\mathcal{T}\mathcal{E}\mathcal{X}$

$\varepsilon\mathcal{T}\mathcal{E}\mathcal{X}$ started out in 1992 as a project by Peter Breitenlohner reimplementing ideas by Knuth [15] for a bi-directional extension but avoiding the need for special DVI drivers. Ideas for additional extensions then were added, and in 1994 the first version of $\varepsilon\mathcal{T}\mathcal{E}\mathcal{X}$ was published.

Around that time members from the $\mathcal{N}\mathcal{T}\mathcal{S}$ team joined the effort and during 1994–98 $\varepsilon\mathcal{T}\mathcal{E}\mathcal{X}$ was run as an $\mathcal{N}\mathcal{T}\mathcal{S}$ -project in order to provide a small number of useful extensions to \TeX to fill the gap while $\mathcal{N}\mathcal{T}\mathcal{S}$ was still under development. As it turned out, however, this set of extensions took on a life of its own and over time was incorporated into all major \TeX -based engines. As a result, nowadays one can assume that all engines support the original \TeX primitives plus the extensions offered by $\varepsilon\mathcal{T}\mathcal{E}\mathcal{X}$.

The new features offered by $\varepsilon\mathcal{T}\mathcal{E}\mathcal{X}$ are a number of additional programming primitives and better tracing facilities, support for mixed-direction typesetting, and an increase in the number of most register types. In the area of micro-typography enhancements, it offers a generalization of `\orphanpenalty` and `\widowpenalty` by supporting special penalty values for the first or last n lines. It also added a method to adjust the spacing in the last line of a para-

graph to be close to that of the preceding line (instead of being set tight as standard \TeX normally does).

2.5 Omega/Aleph

Omega, developed by John Plaice with Yannis Haralambous contributing ideas and developing fonts, was the first extension of the \TeX program that supported Unicode instead of 8-bit input encodings. The driving force behind its development was to enhance \TeX ’s multilingual typesetting abilities and better support for complex scripts.

Aleph is a spin-off of Omega that was started to include $\varepsilon\mathcal{T}\mathcal{E}\mathcal{X}$ capabilities and stabilize the code base. Neither project is being developed any more, but most of Aleph’s and thus Omega’s functionality has been integrated into Lua $\mathcal{T}\mathcal{E}\mathcal{X}$.

2.6 pdf $\mathcal{T}\mathcal{E}\mathcal{X}$

The pdf $\mathcal{T}\mathcal{E}\mathcal{X}$ engine started as a Master’s thesis by Hàn Th   Thành in the mid-nineties and initially offered PDF output, support for embedded Type 1 fonts, virtual fonts, hyper-links, and compression.

For his PhD thesis [37], Hàn Th   Thành experimented with various micro-typography algorithms including the hz approach [42] and several of them were implemented in pdf $\mathcal{T}\mathcal{E}\mathcal{X}$ [38, 39].

Today, pdf $\mathcal{T}\mathcal{E}\mathcal{X}$ (with the $\varepsilon\mathcal{T}\mathcal{E}\mathcal{X}$ extensions included) is the dominant \TeX -based engine in practical use, i.e., all major distributions use this program as the default \TeX engine.

2.7 $\mathcal{X}\mathcal{q}\mathcal{T}\mathcal{E}\mathcal{X}$

$\mathcal{X}\mathcal{q}\mathcal{T}\mathcal{E}\mathcal{X}$ is one of the more recent additions to the \TeX engine successors [6]. It was created by Jonathan Kew and provides as one of its major distinguishing features extensive support for modern font technologies such as OpenType, Graphite and Apple Advanced Typography (AAT). It can make direct use of the advanced typographic features offered by these font technologies, such as alternative glyphs, swashes, optional ligatures, variant weights, etc. These fonts can be used without the need for configuring \TeX font metrics for them.¹

$\mathcal{X}\mathcal{q}\mathcal{T}\mathcal{E}\mathcal{X}$ natively supports Unicode both for input (UTF-8 encoding) as well as for accessing font glyphs. It can also typeset mathematics using Unicode fonts such as Cambria Math or Asana Math [3], provided they contain special mathematical features.

1. The downside of this is that it can’t be guaranteed that the formatting of a source document does not change over time (when libraries are updated on the host system) and there is no way to freeze all components of a document, as is possible with traditional \TeX .

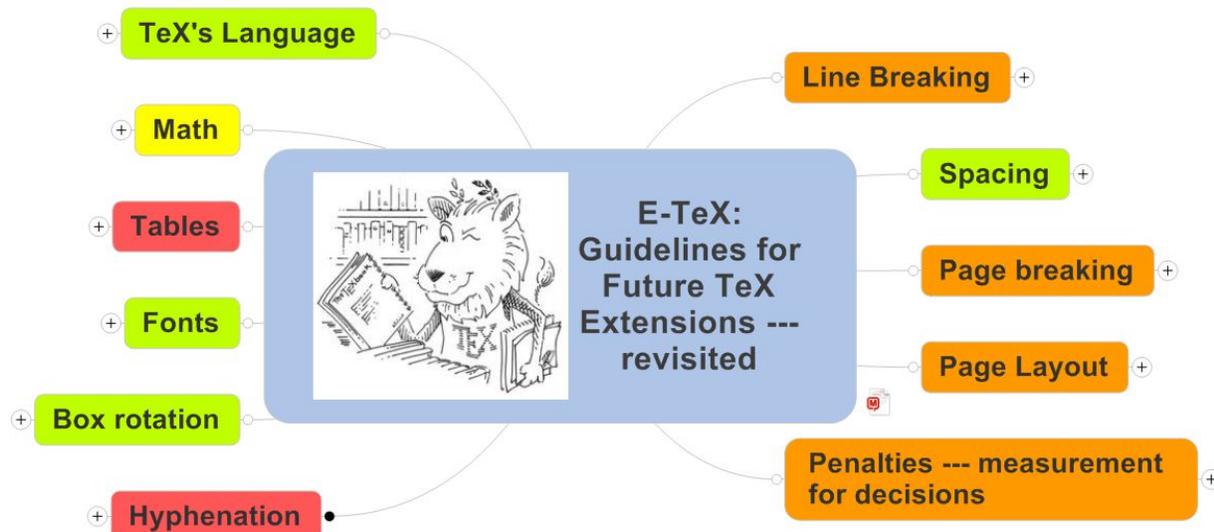


Figure 2: Areas of concern in original article

2.8 LuaTeX

LuaTeX made its first public appearance in 2005 at the TUG conference in China, as a version of pdfTeX with an embedded Lua scripting engine. The first public beta was presented in 2007. It is being developed by a core team of Hans Hagen, Hartmut Henkel and Taco Hoekwater [4].

Important project objectives are merging of engines (combining ideas from Aleph and pdfTeX), support for OpenType fonts, and access to all TeX internals from Lua. Through various callbacks it is possible to hook into TeX's typesetting algorithms and adjust or even replace them.

2.9 iTeX

Finally, as the ultimate successor engine we have or will have iTeX, a fictitious XML-based successor to TeX announced by Donald Knuth at the TUG 2010 conference in San Francisco [14]. According to its author this program will resolve all issues related to high quality typesetting, including those that aren't yet discovered—we can only hope that it doesn't take Don too much time to finish it.

3 Review of the issues raised in 1990

With the knowledge of what today's successors of the TeX engine are capable of, we are now ready to re-analyze the issues discussed two decades ago and evaluate which of them are nowadays:

1. resolved (best case, denoted by  below), or
2. could now be resolved using the improved features of modern engines (hopeful case, denoted by ) , or

3. is still out there waiting for a resolution (bad case, denoted by ).

We follow the order of the original paper (see Figure 2) to help people looking up additional details on the problems outlined. It is available in facsimile on the web [26].

3.1 Line breaking

TeX's line-breaking algorithm is clearly a central part of the TeX system. Instead of finding breaks line by line, the algorithm regards paragraphs as a unit and searches for an 'optimal solution' based on the current values of several parameters. Consequently, a comparison of results produced by TeX and other systems will normally favor TeX's methods.

Such an approach, however, has its drawbacks, especially in situations requiring more than block-style text with a fixed width.

 Issue: No post-processing of final lines based on their content

The final line breaks are determined at a time when information about the content of the current line has been lost (at least for the eyes of TeX, i.e., its own macro language), so that TeX provides no support for post-processing of the final lines based on their content.

For example, the last line of a paragraph is usually typeset using normal spacing between words, *even if the previous line has been set loosely or tightly.* (ϵ -TeX can now handle this to some extent, with its `\lastlinefit` primitive.)

Another example is that the tallest object in a line determines its height or depth so that lines might

get spread apart, *even if they would fit perfectly*.

In theory these issues could now be catered to with the LuaTeX program, because it offers the ability to post-process the lines and modify the final appearance.

  **Issue:** No way to influence the paragraph shape with regard to the current position on the page

TeX and all its successors break paragraph text into lines at a time where they do not know where this text will eventually appear on a page. Consequently, there is no possibility within the model of catering to special paragraph shape requirements based on that position.

The only way to work around this is a complex feedback loop, using placement information from a previous run to calculate the necessary `\parshape`. Because this requires multi-pass formatting (with many passes), it is impractical. A simpler, though still complicated, approach is to assume a strict linear formatting, in which case one can build the paragraph shapes one after the other.

A new approach, that we are currently exploring for L^ATeX3, involves storing paragraph data in a data structure that allows re-breaking the material for trial typesetting. This is outlined in Section 4.

3.1.1 Line-breaking parameters

While the algorithm provides a wide variety of parameters to influence layout, some important ones for high-quality typesetting are missing. To resolve some of these issues, we need only (slightly) modify or extend the current algorithm. For others, serious research is required just to understand how a solution might be approached.

None of the engines has modified the TeX algorithm, so all of the problems are still unsolved. LuaTeX offers a way to replace the whole algorithm, but for most of these problems, that would be overkill, because it would require reprogramming everything from scratch in Lua.

 **Issue:** Zero-width indentation box

When TeX breaks text into individual lines it discards whitespace and kerns at both sides of each line except for the first. On the left side of the first line (in left-to-right formatting) the existence of the paragraph indentation box prevents this from happening. Normally this is not noticeable, but in the case of layouts without paragraph indentation it can lead to problems, e.g., when `\mathsurround` has a positive value.

In LuaTeX this could now be resolved by defining code that preprocesses the paragraph material

and removes discardable items following the indentation box.

 **Issue:** Managing consecutive hyphens in a general way

In TeX it is possible to discourage two consecutive hyphens, but there is no way to prohibit or strongly discourage three or more. Technically, this would mean a slight extension of the current algorithm by keeping track of the number of hyphens in a row. None of today's engines supports that concept.

 **Issue:** Only four types of line quality

To implement good-looking paragraphs, TeX classifies each line into one of four categories based on the line's glue setting (*tight*, *decent*, *loose*, *very loose*). It then uses that classification to avoid abrupt changes in spacing (if possible). However, the small number of classes results in grouping of fairly incompatible settings in a single class (especially, *loose* and *very loose* are affected). Technically, it would be simple to extend the number of classes to support better granularity.

 **Issue:** Rivers and identical words across lines

If interword spaces from different lines happen to fall close to each other, they form noticeable stripes (*rivers*) through the paragraph that can be quite disconcerting. TeX's line-breaking algorithm is unable to detect such situations. Resolving this would require serious research into the question on how to detect rivers and how to classify the "badness" of different scenarios in order to programmatically handle it through an algorithm.

A somewhat related issue (but rather easier to resolve) is the placement of the same word at the same position in consecutive lines, especially at the beginnings of lines, which is likely to disrupt the reading flow.

3.2 Spacing

Micro-typography deals with methods for improving the readability and appearance of text; see for example [5]. While TeX already does a great job in this area, some of the finer controls and methods are not available in the original program.

However, most of them have been implemented in some of the successor engines and an interface for L^ATeX to these micro-typography features is provided through the package `microtype` [33].

 **Issue:** No flexible interword spacing

In order to produce justified text, a line-breaking algorithm has to stretch or shrink the interword

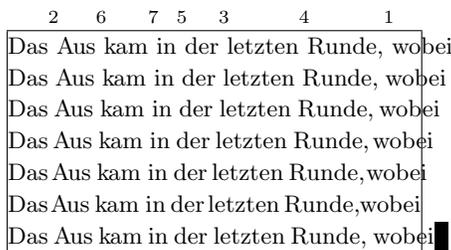


Figure 3: Interword spacing

The interword spaces are numbered in a way so that higher numbers denote spaces which should shrink less using the rules given by Siemoneit [34]. The last line shows the resulting overflow box which would be produced by standard \TeX in this situation.

space starting from some optimal value (e.g., given by the font designer) until the final word positions are determined. \TeX has a well-designed algorithm to take such stretchability into account. It can also alter spacing depending on the character in front of the space to change the behavior after punctuation, for example.

There is no provision, however, for influencing the interword spaces in relation to the current characters on both word boundaries. Ideally, shrinking or stretching should depend on the character shapes on both sides of the space as exemplified in Figure 3.

None of the \TeX successors provides any additional support for controlling the interword spacing above and beyond \TeX 's capabilities. But with Lua \TeX 's callback interfaces it is possible to analyze and modify textual material just before it is passed to the line-breaking algorithm. This allows for ways to resolve this issue either as a table-based solution (one size fits all), or on a more granular level where the chosen adjustments are tied to the current font.

👉 Issue: No flexible intercharacter spacing

Instead of, or in addition to, stretching or shrinking the interword spaces to produce justified text, there are also the methods of *tracking* (increasing or decreasing inter-letter spaces) and *expansion* (changing the width of glyphs). There are debates by designers whether such distortions are acceptable approaches, but there is not much doubt that, if used with care and not excessively, they can help to successfully resolve difficult typesetting scenarios.²

pdf \TeX provides both methods, the latter by implementing a version of the hz algorithm originally developed by Hermann Zapf and Peter Karow [42].

👉 Issue: No native support for hanging punctuation

Don Knuth [18, pp. 394–395] gave an example of how to achieve hanging punctuation but it required the

use of specially adjusted fonts and it also interfered with the ligature mechanism. In other words, it is only a partial solution for restricted scenarios.

Fortunately, a fully general solution was implemented in pdf \TeX and later also incorporated into Lua \TeX , so nowadays this can be considered resolved. *The remainder of the article is typeset using hanging punctuation to allow for a comparison.*

3.3 Page breaking

In 1990 I wrote “The main contribution of \TeX 82 to computer-based typesetting was the step taken from a line-by-line paragraph-breaking algorithm to a global optimizing algorithm. The main goal for a future system should be to solve the similar, but more complex, problem of global page breaking”. Unfortunately in the \TeX world no serious attempt was made since then to address the fundamental limitation in \TeX 's algorithm, let alone designing and implementing a globally optimizing page-breaking algorithm.³

👉 Issue: \TeX generates pages based on precompiled paragraph data

This issue describes the fundamental problem in \TeX 's approach: the program builds optimized paragraph shapes without any knowledge about their final placement on a page. The result is a “galley” from which columns are cut to a specified vertical size. A consequence of this is that one can't have the shape of a paragraph depend on its final position on the page when using \TeX 's page builder algorithm.

To some extent, it is possible to program around this limitation, e.g., by measuring the remaining space on a page and explicitly changing paragraph shapes after determining where the current textual material will finally appear. However, besides being complicated to implement, it requires accounting for all kinds of special situations that normally would be automatically managed by \TeX , and providing “programmed” solutions for them.

As a result, all attempts so far to provide such functionality had to impose strong limitations on the allowed input material, i.e., they worked only in restricted setups and even then, the results were often not satisfactory.

2. On page 54 one paragraph was typeset with a negative expansion of 3% to avoid an overflow line. See if you can spot it without peeking at the end of the article where we reveal which it was.

3. In the wider document engineering research community some research was carried out in the last thirty years, e.g., [7, 9, 22, 41], but so far none has led to a production system.

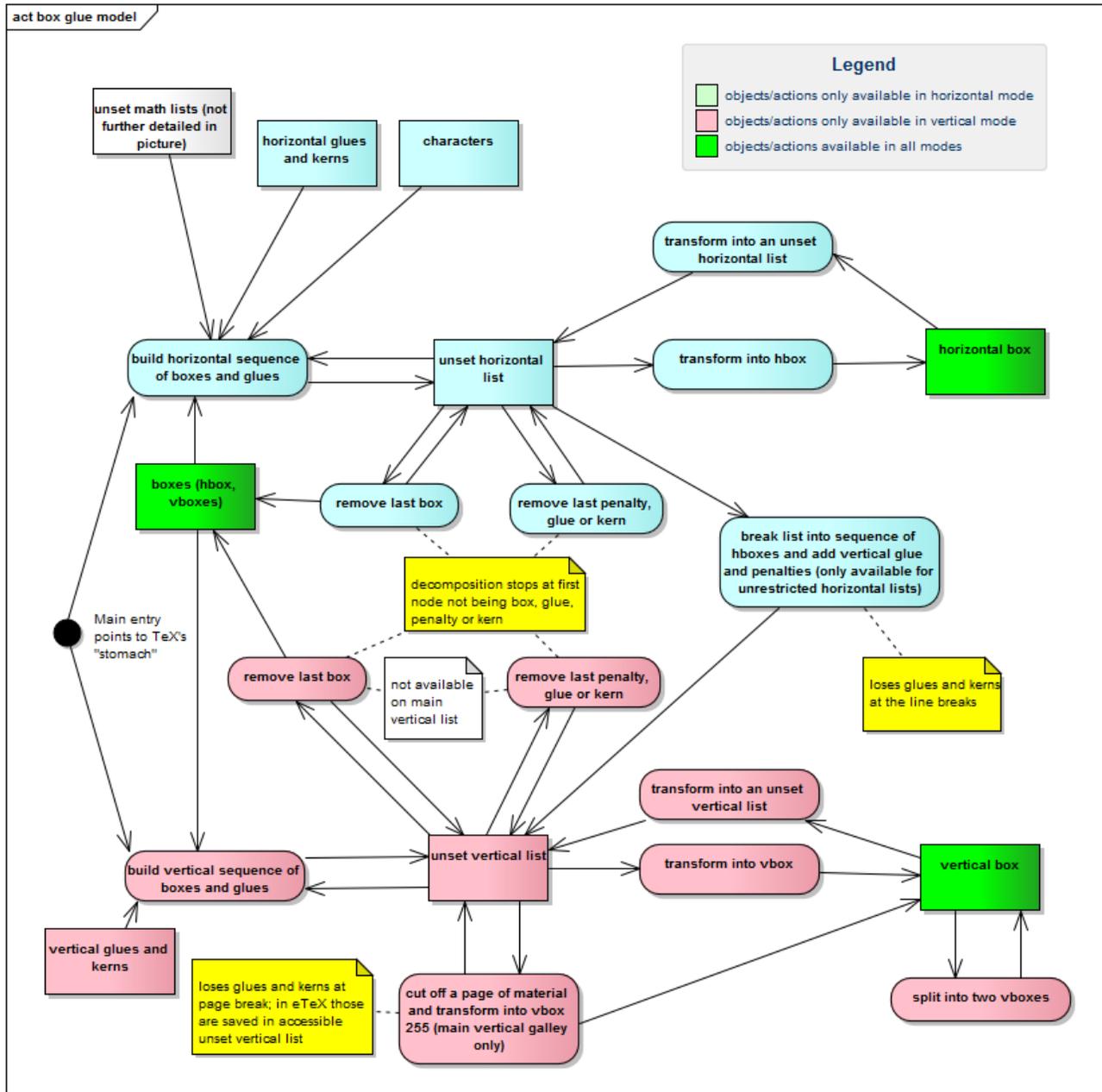


Figure 4: TeX’s box/glue/penalty model

👉👈 Issue: Paragraphs already broken into columns can’t be reformatted based on page/column break decisions

The main operations possible in TeX’s box/glue/penalty-model are shown in Figure 4. All macro processing that acts on the level of tokens (characters/symbols, spaces, etc.) is only possible before TeX builds the so-called “unset horizontal lists” in which character tokens change their nature into glyphs from fonts. From that point on, the manipulation possibilities are reduced to the level of box manipu-

lations that only allow relatively few actions, such as removal of the last item in a box. However, those operations have severe limitations, e.g., one can’t remove glyphs from a horizontal list nor is there any possibility to convert the data back to character tokens, etc. that could be directly reprocessed by the macro processor.

The moment TeX turns an “unset horizontal list” into an “unset vertical list”, i.e., when it applies line breaking, we move to the bottom half of the model and from there, there is no fully general way

to get back to the upper half. At the line breaks, we potentially lose spaces that can't be recovered. Thus, it is not possible to reconstruct the original “unset horizontal list” even if we would recursively take off items from the end of the “unset vertical list” in the attempt to reassemble it.

As a consequence it is not possible to safely reuse textual material once it has been manipulated by \TeX 's paragraph builder. Instead one needs to find a way to record the “unset horizontal list”. That this is easily possible in $\text{Lua}\TeX$ (but also in standard \TeX with somewhat more effort) will be demonstrated in Section 4 on page 59, which is the reason why we give this issue a combined  rating.

3.4 Page layout

For the tasks of page makeup, \TeX provides the concept of output routines together with insertions and marks. The concepts of insertions and marks are tailored to the needs of a relatively simple page layout model involving only one column output, footnotes, and at the most, simple figures once in a while.⁴

The mark mechanism provides information about certain objects and their relative order on the current page, or more specifically, information about the first and last of these objects on the current page and about the last of these objects on any of the preceding pages. However, being a global concept only one class of objects can take advantage of the whole mechanism.⁵

 Issue: Only a single type of marks is fully supported

In the original paper, I suggested extending this to support multiple independent mark registers; that idea was later implemented in the $\varepsilon\text{-}\TeX$ program. As it turned out, however, this did not really solve the issue. Whenever the page layout gets more complicated and output routines are used to inspect the current state without actually shipping out pages in a linear fashion, the information maintained in `\topmark` is always lost.

In the end, we abandoned the whole mechanism for $\text{\LaTeX}3$ and used only \TeX 's `\botmark` register to put marks on the page and kept track of all other information (class of mark and content of the mark) externally [27]. This solves the issue, but at a fairly high programming cost with complex data management.

 Issue: Missing built-in support for complex float management

Float placement across different types of publications is governed by rules of high complexity: placement

options may depend on aesthetic requirements, captions and legends might require different formatting depending on placement, positioning of floats may influence options available for other floats, etc.

Unfortunately, out of the box, \TeX offers only a simplistic mechanism derived as an extension to the footnote concept. \LaTeX extended this to a slightly more flexible algorithm but only with respect to supporting different classes of floats (where the floats of one class have to stay in sequence) and by adding a few parameters to add limits for the number of floats in a float area or the maximum size of an area. More complex rules or arrangements with varying formatting depending on placements, support for floats across multiple columns (other than a simple two-column mode) are not supported.

To some extent, this is not that surprising, because codification of placement rules and effective algorithms for computing complex layouts is an area that is not well understood, and at the same time hasn't attracted much attention by the research community. Only a handful of publications in three decades approach one or another aspect of this topic [7, 12, 25, 27, 30]. For the same reason, none of the newer engines offers any additional built-in support that would ease the implementation of more complex algorithms. Using an early version of `expl3`, an attempt was made [27] to design and implement a customizable float algorithm that supports a richer set of rules. Unfortunately this has not yet gone beyond a proof-of-concept implementation.⁶

 Issue: No conceptual support for baseline to baseline spacing

For designers, \TeX 's way of specifying interline glue is a rather foreign concept; they typically use baseline-to-baseline spacing instructions in their specifications. Unfortunately, those prescriptions are not directly possible in \TeX because of the way \TeX determines the “current” `\baselineskip` value: see Figure 5 on the next page. Only with rigorous control on the

4. The term ‘one column output’ means that all text is assembled using the same line width. Problems with variable line width are discussed in Section 3.3. Of course, this already covers a wide range of possible multi-column layouts, e.g., the footnote handling in this article. But a similar range of interesting layouts is not definable in \TeX 's box-glue-penalty model.

5. The \LaTeX implementation provides an extended mark mechanism with two kinds of independent marks with the result that one always behaves like a `\firstmark` and the other like a `\botmark`. The information contained in the primitive `\topmark` is lost.

6. A new implementation of these ideas using the current `expl3` language is high on the current $\text{\LaTeX}3$ road map.

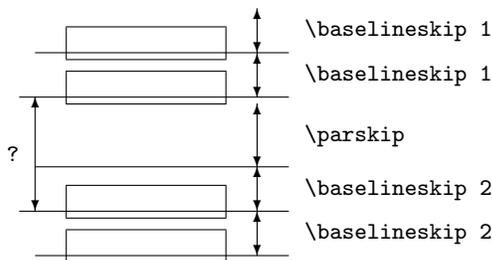


Figure 5: Baseline to baseline spacing

To implement a baseline to baseline dimension, for example between a paragraph and a heading (denoted by the question mark), the value for `\parskip` has to be determined depending on the `\baselineskip` of the second paragraph. Unfortunately, the value of `\baselineskip` used will be the one current at the end of the second paragraph while the `\parskip` has to be computed at its beginning.

programming level (i.e., by preventing the users and package writer from accessing the full power of `TEX`) can one provide an interface supporting baseline-to-baseline spacing specification.

Because the interline glue concept is deeply buried in `TEX` algorithms, it is not surprising that none of today’s engines (including `LuaTEX`) addresses this area.

Issue: No built-in support for grid-based design

`TEX`’s concepts that conflict with baseline to baseline spacing also make grid-based design difficult, as this strongly depends on text baselines appearing in predictable positions.

Nevertheless, over the last three decades there have been a number of attempts to provide support for grid based-design on top of standard `LATEX`. None of them has been particularly successful — due to the missing support from the engine, all of them worked only with subsets of the many `LATEX` packages. To make things work, it is necessary to adjust behavior of various commands, and thus any package not explicitly considered is a likely candidate for disaster.

The `xor` package for `LATEX3` offers a structured interface on the code level for grid-design. Unfortunately, it isn’t production ready and awaits a major refactoring based on the new `expl3` language for `LATEX3`. But even then it would suffer from the limitations listed above as long as it is used together with `LATEX2ε` packages that do not interface with its grid support.

From an engine perspective, `LuaTEX` is the only engine that offers some additional possibilities that may help with grid design, through additional hooks provided, and through access to the internal `TEX`

data structures. If and how this could be usefully deployed remains to be seen. To my knowledge, no `LuaTEX`-specific code for grid design yet exists.

3.5 Penalties — measurement for decisions

Line and page breaks in `TEX` are determined chiefly by weighing the “badness” of the resulting output⁷ and the penalty for breaking at the current point. This works very well in most situations but there is one severe problem with the concept of penalties.

Issue: Consecutive penalties behave as $\min(p_1, p_2)$

The problem is that an implicit penalty (e.g., for discouraging but not prohibiting orphans or widows) will always allow a break at this particular point even if an explicit penalty by the user attempts to disallow a break there. Changing the algorithm to use $\max(p_1, p_2)$ instead would resolve the problem. With this approach an explicit breakpoint could still be inserted by interrupting the sequence of consecutive penalties, e.g., through `\kern0pt`.

With `LuaTEX` this could probably be implemented, but would most likely require very complicated parsing and manipulation of internal `TEX` data structures, unless `LuaTEX` itself gets extended, i.e., by making the code that handles consecutive penalties directly accessible.

3.6 Hyphenation

When typesetting text, especially in narrow columns, hyphenation is often inevitable in order to avoid unreadable, spaced out lines. `TEX`’s pattern-based hyphenation algorithm [23] is quite good at identifying correct hyphenation points to avoid such situations.

However, hyphenation is a second-best solution and, if applied, there are a number of guidelines that an algorithm should follow to improve the overall result. Several of them cannot be specified with `TEX`’s algorithm. In fact, all of the ones below are unresolved with today’s engines, if we disregard that in `LuaTEX` one could in principle replace the full paragraph-breaking algorithm with a new routine.

Issue: Prevent more than two consecutive hyphens

Hyphenation of two consecutive lines is controlled by the algorithm (`\doublehyphendemerits`),

⁷ The badness is a function of the white space used in a line in comparison to the optimal amount, e.g., if the space between words needs to stretch or shrink, the badness of the solution increases.

but there is no possibility of avoiding paragraphs like the current one, in certain circumstances. As one can easily observe, the number of hyphens in this paragraph has been artificially increased by setting relevant line-breaking parameters to unusual values. In languages that have longer average word lengths than English, such situations present real-life problems.

 Issue: Assigning weights to hyphenation points

\TeX 's hyphenation algorithm knows only two states: a place in a word can or cannot act as a hyphenation point. However, in real life certain hyphenation points are preferable over others. For example, “Nonnenkloster” (abbey of nuns) should preferably not be hyphenated as “Nonnenklo-ster” (as that would leave the word “nun’s toilet” on the first line).

 Issue: More generally, support other approaches to hyphenation

Liang’s pattern-based approach works very well for languages for which the hyphenation rules can be expressed as patterns of adjacent characters next to hyphenation points. Such patterns may not be easy to detect but once determined they will hyphenate reasonably well. For the approach to be usable, the necessary set of patterns should be reasonably small, as each discrepancy needs one or more exception patterns with the result that the pattern set would either become very large or the hyphenation results would have many errors.

To improve the situation for the latter type of languages one would need to implement and potentially first develop other types of approaches. For now Liang’s algorithm is hardwired in all engines, though in theory \LuaTeX offers possibilities of dropping in some replacement.

3.7 Box rotation

\TeX 's concept of document representation is strictly horizontal and left-to-right oriented. Any further manipulation is left to the capabilities of the output device using \special commands in the language of the device.

 Issue: No built-in support for rotation

Because of the lack of a common interface for such operations, any document making use of \special commands is processable only in a specific environment, so that exchange of such documents is only possible in a limited fashion.

With the event of $\text{\LaTeX} 2_{\epsilon}$ the \LaTeX project team resolved this issue by providing an abstract interface layer that (in the form of the `graphics` and

`color` packages) hides the device peculiarities internally so that documents using these interfaces became portable again.

3.8 Fonts

 Issue: Available font information (non-issue)

In the Texas paper, I suggested that additional font characteristics should be made available as font parameters to enable smarter layout algorithms. Looking at this from today’s perspective I think it was largely a misguided idea, at least until recently. Many of the fonts that have been made accessible to the \TeX engines in the last decades (mainly PostScript Type 1 fonts) do indeed have various additional attributes defined by their designers but alas with largely non-comparable values between different fonts making any interpretation difficult if not impossible.

With OpenType fonts, this may change again, and engines like \XeTeX or \LuaTeX allow access to such additional attributes.

  Issue: Encoding standardization

By default, \TeX translates the input encoding (representation of characters in the document) one to one into the output encoding (glyph positions in the current font). E.g., if you put a “b” into your document then this is understood as “typeset the character in position 96 (ASCII code for “b”) in the current font”. This tight coupling between encoding of data in different places required that fonts used by \TeX always stored the glyphs in the same position (which they did only partially) or that the user understood the subtle differences and adjusted the document input accordingly. For example, in plain \TeX the command $\text{\$}$ produces a $\text{\$}$ -sign — unless you are typesetting in *Computer Modern Text Italic*, in which case your output suddenly shows $\text{\mathcal{L}}$ -signs.

With more and more fonts (with different font encodings) becoming available and \TeX entering the 8-bit world (with numerous input encodings interpreting the document source differently), such issues got worse.

These problems were resolved for \LaTeX through three developments. Don Knuth developed the idea of virtual fonts [13] and that concept was quickly adopted by nearly all major output-device drivers, so that it became usable in practical terms.⁸ With this concept available the \TeX community agreed in Cork on a special virtual font encoding [2]. Finally we designed and implemented the \LaTeX Internal Char-

8. In 1990 I expressed my hope that these ideas would help to simplify matters [26, p. 342] and as it turned out, that was indeed the case.

acter Encoding for L^AT_EX 2_ε (LICR) [29, chap. 7] that transparently maps between different input encodings and arbitrary font encodings. This is perhaps not a perfect solution, but for the 8-bit world it effectively resolved the issues.

Unicode support was also addressed (through the `inputenc` package) but here better support from the engines is required to come to a fully satisfactory solution. As mentioned above, explicit Unicode support was first added by Omega, and from there made its way into Aleph and LuaT_EX. X_ƒT_EX also natively supports Unicode.

👉 Issue: Ligature and kerning table manipulation

In T_EX, ligatures and kerns are properties of fonts, i.e., they apply to all text in a document. However, different languages use different rules about what to apply or not to apply in this case. Thus to model this in T_EX, one would need to define private fonts per language, each differing only in their ligature and kerning tables. While this would be a theoretical possibility, in practical terms it would be a logistical nightmare and so nobody has ever tried to implement such fine points of micro-typography.

With pdfT_EX this situation changed somewhat as pdfT_EX supports suppressing all ligatures or all ligatures that start with a certain character. This alone does not help much though, as it does not allow, for example, prohibiting the “ffl” ligature (not used in the German language) while allowing for the other ligatures starting with “f”, nor does it support implementing new ligatures, such as “ft” or “ck”, through negative kerning.

LuaT_EX takes this a huge step forward and provides the necessary controls to improve the situation considerably. While I was writing this article (and asking around), the first experimental packages started to appear, e.g., `selnolig` [24] by Mico Loretan, so it will be interesting to see what happens in the near future.

3.9 Tables

👉 Issue: Combining horizontally- and vertically-spanned columns is impossible

T_EX’s input format is inherently linear, and so it is not surprising that any T_EX interface to inherently two-dimensional table data is somewhat limited. Out of the box T_EX supports column formatting, e.g., it can calculate the necessary column width and apply a default formatting per column. It also allows for horizontally-spanned cells with their own formatting. However, there is no provision for providing cells whose content is able to reflow depending on the available space nor is there any mechanism to

provide vertically-spanned cells. Both are essential formatting requirements.

L^AT_EX offers a higher-level document syntax to the low-level capabilities that T_EX provides, and over time, many packages appeared that enhanced the solution in one way or the other. However, without any underlying direct support for the more complex concepts all these efforts show limitations and are often difficult to use. So far none of the existing engines addresses this area.

3.10 Math

Mathematical typesetting is one of T_EX’s major domains where—even today after thirty years—no other automatic typesetting system has been able fully to catch up. But even in this area several things could be improved.

👉👉 Issue: Some mathematical constructs are not naturally available in T_EX, e.g., double accents, under-accents, equation number placement, ...

For many of these problems workarounds have been implemented as (fairly complex) plain T_EX macros by Michael Spivak in the $\mathcal{A}\mathcal{M}\mathcal{S}$ -T_EX format [35, 36]. Most of this code plus further extensions were ported to L^AT_EX and are nowadays available as the L^AT_EX package `amsmath`.

For this reason this issue can be largely regarded as solved, even though native support for most of these constructs would improve the situation further.

👉 Issue: Spacing rules and parameters are all hardwired in the engine or the math fonts

T_EX’s spacing rules for math are quite good, but in cases where they needed adjustments, it was either impossible or quite difficult to do. This restriction has been finally lifted with LuaT_EX, because that engine offers access to all internal parameters of T_EX.

👉 Issue: Sub-formulas are always typeset at their natural width

No engine so far provided any alteration to the core algorithms of T_EX that format a formula. Thus sub-formulas (for example from `\left ... \right`) are still boxed at natural width even if the top level math-list is subject to stretching or shrinking. This also means that there is no way to automatically break such constructs across lines.

👉 Issue: Line breaking in formulas (not listed in original paper)

Don declared line breaking in math too hard for T_EX to do automatically and as a result countless users struggled with manually formatting displayed

equations to fit a given measure. For a long time it looked as if that problem was indeed too difficult to tackle within the constraints of a formatting engine. However, in the late nineties Michael Downes proved everybody wrong by designing and implementing a first version of the `breqn` package for L^AT_EX.

This package—further improved by Morten Høgholm after Michael’s untimely death—gets us already quite a way toward the goal of high-quality automatic line breaking in formulas [10]. However, with the increased processing power now available and with LuaT_EX’s access to T_EX internals, it should be possible to solve most or all of the remaining problems identified.

3.11 T_EX’s language

In 1990 I made the bold statement: “T_EX’s language is suitable for simple programming jobs. It is like the step taken from machine language to assembler”. Since then the new engines added one or the other primitive, but with the exception of LuaT_EX, which added an interface to Lua as an additional programming language, the situation hasn’t improved with respect to core engine support.

👍 Issue: Incompleteness with respect to standard programming constructs

While this statement is still true with respect to most engines, the situation has nevertheless improved. With `expl3` the L^AT_EX Project team has provided a programming layer that offers a large set of data types and programming constructs. The development of `expl3` started in fact long ago: initial implementations date back to 1992. However, back then, the processing power of machines was not good enough to allow executing code written in `expl3` at a reasonable speed. For that reason, the ideas and concepts were put on the shelf, and the project team instead concentrated on providing, and later on maintaining, L^AT_EX 2_ε.

Since then processor speed has increased tremendously, and as a result it became feasible to finally use the ideas that had been initially developed nearly two decades ago. The core of `expl3` has been reimplemented (again) and its stable release is now gaining more and more friends—it even got its own logo designed as shown in Figure 6.

👍 Issue: A macro language ... good or bad? / Difficulty of managing expansion-based code

Since the first implementation of T_EX people have voiced their concern about the fact that the T_EX language is a macro language that works by expansion and not a “proper” programming language. However,



Figure 6: The `expl3` logo (courtesy of Paulo Cereda)

despite this grumbling, nobody came up with a workable alternative model that successfully combines need for simple and easy input of document material (which makes up the bulk of a T_EX document) and the special needs of a programming environment that avoids the complexity of programming by macro expansion (which indeed becomes complex and difficult to understand if used for non-trivial tasks).

That the T_EX language can be used to produce truly obfuscated code was nicely demonstrated by David Carlisle’s seasonal puzzle [8] which is worth taking a look at, but even normal coding practice will easily lead to code that is difficult to understand (and to maintain) as demonstrated by many of today’s packages. Part of the reason for this is that all coding practice around L^AT_EX 2_ε (and other macro formats) is based on concepts and ideas originated in plain T_EX, with more and more complexity layered on top but without fundamentally questioning the core approach which was never intended for complex programming tasks.

So why 👍? Largely because with `expl3` we now have a foundation layer available, that—while still based on macro expansion—provides a comfortable programming environment. From the engine side LuaT_EX nicely sidesteps the question by providing a separate programming language in addition.

👍 Issue: Inability to access and manipulate certain internal T_EX data structures

Many of T_EX’s internal data structures are inaccessible to the programmer or only observable indirectly.⁹ Thus, whenever an adjustment to one of T_EX’s internal algorithms is needed it becomes necessary to bypass the algorithm completely and reimplement it on the macro level. This means accepting huge inefficiencies and in many cases makes such implementations unusable in real-life applications.

9. In the 1990 paper I gave the example of measuring the length of the last line in a paragraph by artificially following it by an invisible displayed formula as only within such a display is the desired information made available.

For more than two decades this was the situation with all engines. Finally LuaTeX broke this restriction by offering access to all (or nearly all) internals including the ability to modify them (with the danger of breaking the system in unforeseen ways, but that comes with the territory).

👍👎 Issue: The problem of mouth and stomach separation

This is perhaps one of most fundamental issues: not so much with the language but with the underlying data structure. A special case of this issue — and perhaps the most important one — was already discussed in Section 3.3 on page 53: the inability to reformat paragraph data that is already broken into individual lines.

TeX divides its internal processing into two parts: the token parsing and manipulation where expansion happens (termed the “mouth”) and the box generation and manipulation processes that build up the elements on the page (called the “stomach”). Figure 4 on page 53 depicts the operations available in the “stomach” with the two main entry points from the “mouth” on the far left. And, as in real life, this is largely a one-way street, i.e., once tokens have been transformed into boxes and glue there is no way of getting back to tokens. Furthermore, manipulation possibilities of already-processed material are limited and usually result in loss of information, so that one has to ensure staying at the token level until the very last moment.

Unfortunately there are also issues with staying at the token level. For one, only the typesetting stage will provide the necessary information to successfully position material on the page, e.g., to find out how much space some text will occupy or where and on which page a reference to a float will fall. Thus trial typesetting is necessary and the source material would need to be stored on the token level.

However, reprocessing token material means that the same macro processing happens several times when you are doing the trial typesetting. If this processing has side effects, such as incrementing a counter, one needs to keep track of all such changes and undo them before restarting a trial.

From the engine perspective the best approach would be to either

- provide access and manipulation possibilities from the “mouth” to an intermediate data structure that holds character data plus attributes before they are turned into glyphs, or
- provide additional manipulation possibilities of “stomach” material, or

- offer conversion of typeset material back to token data similar to what `\showbox` offers as symbolic information in the transcript file.

Sadly, none of the engines offers any direct support in this area. However, we will see that, with today’s increase in processing power, it becomes feasible to implement a strictly TeX-based solution. This solution has some (acceptable) limitations for boundary cases, but a variant implementation using LuaTeX’s callback interface can even get rid of those, as we’ll see in the next section. For this reason we give this issue a 👍👎 rating today.

4 Overcoming the mouth/stomach separation

If we look back to Figure 4 on page 53 we can see that the best data structure available for use in trial typesetting is the “unset horizontal list”. The moment we apply line breaking we would lose information and if we store the information at an earlier stage (i.e., as token data) we would have to deal with the side effects of repetitive token processing.

Unfortunately, the “unset horizontal list” is not a data structure made available by TeX. What is possible though (looking at the right side of the diagram), is to store it in a horizontal box. At a later stage this box can then be transformed back into an “unset horizontal list” and that could then be typeset into a “trial” paragraph shape.

However, simply storing the content of one or more paragraphs into an `\hbox` for later processing is not a workable option either, because:

- TeX applies some “optimizations” in restricted horizontal mode to save some processing time.¹⁰ Under the assumption that text in an `\hbox` cannot be broken into several lines, it drops all break penalties from in-line formulas (i.e., those normally added after binaries and relational symbols) and also doesn’t add any implicit `\discretionary` hyphens in place of “-”. For the same reason it also ignores changes to `\language`.
- If we save each paragraph into one `\hbox` then we effectively surround each paragraph by a TeX group. Thus local changes, such as modifications to fonts or language would suddenly end at the paragraph boundary.

While these restrictions can be overcome, it means a far more elaborate approach needs to be taken.

10. While such optimizations have been important at the time TeX was originally developed, the speed gain they offer nowadays is negligible. What remains are inconsistencies in processing that should get removed in today’s engines such as pdfTeX and LuaTeX.

4.1 A standard TeX solution

If storing the “unset horizontal list” directly in an `\hbox` is not an option, what alternative is available according to our Figure 4? The only other path that results in an `\hbox` is to transform the list into an “unset vertical list” and then remove the last box from that list (i.e., a circular path in clockwise direction around the center of the diagram). To make this work we have to overcome the limitations listed at various places along the path:

1. Transforming an “unset horizontal list” into an “unset vertical list” loses glues and penalties at line breaks.
2. Decomposition of the “unset vertical list” is not possible on the main galley.
3. Decomposition of the “unset vertical list” stops at the first node that is not a box, glue, penalty, or kern item.

To alleviate issue 1 we will build our “unset vertical list” using the largest possible `\hsize` available in TeX. We remove the indentation box whenever we start a paragraph. In addition, we trap any forced break penalty, record it, and prematurely end the paragraph at this point to stay in control. As a result each `\hbox` in the resulting “unset vertical list” will contain exactly the paragraph material from one forced break to the next (considering the paragraph boundaries as forced breaks).

What happens if the paragraph material exceeds the largest possible dimension available in TeX? In that case (which means that the paragraph is noticeably longer than a page) we will end up with uncontrolled line breaks. In TeX it is impossible to prevent this from happening, but at least it is possible to detect that it happened. One can then warn the user and request that the paragraph be artificially split somewhere in the middle using a special command.

Issue 2 is easily resolved: as we initially want to store the data, we can simply scan the material within a `\vbox`. This box, which is later thrown away, will form a boundary for local changes, but this is okay, as we can scan as many paragraphs as necessary in one go.¹¹ This would solve the problem if only portions of a document (e.g., float captions) are subject to trial typesetting. If the intention is to process the whole document in this manner then a slightly different approach is needed. In that case we would use the main vertical list for collection and devise a special output routine that is triggered at the end of each paragraph. The `\hboxes` holding the paragraph fragments would then be retrieved within the output routine. Once everything is collected as

far as necessary, the output routine could then be changed to do trial typesetting.

To avoid issue 3 it is necessary to ensure that material from `\insert` and `\adjust` is not migrated out of the horizontal material. If they were, they would appear after our “paragraph line”. On the one hand, this is the wrong place if we later rebreak the material into several lines, and on the other hand it would prevent us from disassembling the material and storing it away. Therefore the solution is to end the paragraph (generating one line for the current fragment that we can save away), then start an `\hbox` into which we scan the `\insert` or `\adjust` and then restart the scanning for the remainder of the “real” paragraph.

With these preparations, the algorithm then works as follows:

- When we start (or restart) scanning paragraph material we ensure that there is no indentation box at the beginning.
- When we reach the end of the paragraph (or a paragraph fragment where we have artificially forced a paragraph end) TeX will typeset the material and because of the large line length it will (normally) result in a single-line paragraph. We then pick up this line via `\lastbox` and repackage it by removing the glue (from `\parfillskip`) and penalty at its end. Then we save this box and an accessing function away in some data structure and restart scanning until we reach the end.
- If we see an `\insert` or `\adjust` we interrupt and add the scanned material to the data structure. Then scan and store the vertical material as outlined above.
- If we see a forced penalty we interrupt and save the scanned material and then also record the value of the penalty in the data structure. Note that any non-forcing penalty could just be scanned as normal paragraph material because of the large `\hsize`.
- Once we are finished parsing we end up with a data structure that looks conceptually as follows:

```
\dobox box1 \dopenalty {10000}
\dobox box2 \doinsert boxx
\dobox box3 \doadjust boxy ...
```

box₁ to *box₃* are `\hboxes` holding paragraph text fragments; *box_x* and *box_y* are also `\hboxes` containing just an `\insert` or `\adjust`, respectively, i.e., they are generated basically by:

```
\setbox boxx =\hbox{\insert{...}}
```

11. The limit is available memory, which is huge these days.

With the right definitions for `\dobox` and friends (e.g., `\unhcopy`, etc.) this data structure can then be used to “pour” the saved paragraph(s) into various molds for trial typesetting.

This algorithm works with any T_EX engine and its only restriction is the maximum allowed size of a single paragraph. This is acceptable, as it normally would not happen unless somebody is typesetting a document in James Joyce style. If it does, it will be detected and the user can be asked to artificially split the paragraph at a suitable point.

4.2 A LuaT_EX solution

Using the LuaT_EX engine, it is possible to simplify the algorithm considerably. LuaT_EX offers the possibility of replacing the line breaking algorithm with arbitrary Lua code and we can use this fact to temporarily replace it with a very trivial function: one that simply packages the “unset horizontal list” into an `\hbox` and returns that. This would look as follows:

```
function hpack_paragraph (head)
  local h = node.hpack(head)
  return h
end
callback.register("linebreak_filter",
                  hpack_paragraph)
```

The beauty of this is that it automatically resolves issues 1 and 3 listed above. Issue 1 is fully resolved, because `node.hpack` is able to build `\hboxes` of any size, even wider than `\maxdimen` (as long as you do not try to access the `\wd` of the resulting box). So even Joycean paragraphs are no longer any problem. Issue 3 is gone because we do not need to decompose material. With the simple code above any penalties, `\inserts`, or `\adjusts` simply end up within the box; in contrast to the normal line breaking algorithm, the `hpack_paragraph` code does not touch them. For the same reason, we do not have to take off any `\parfillskip` from the end, as it isn’t added in the first place.

The only problem found so far is a bug in the current implementation of the `linebreak_filter` callback rendering `\lastbox` (which is needed by our algorithm) unusable the moment the callback is installed. This is due to some missing settings in the semantic nest of T_EX that are not fully carried out. Eventually this will most certainly get corrected in a future LuaT_EX version. For now it is possible to implement the correction ourselves in a second callback:

```
function fix_nest_tail (head)
  tex.nest[tex.nest.ptr].tail = node.tail(head)
  tex.nest[tex.nest.ptr].prevgraf = 1
```

```
tex.nest[tex.nest.ptr].prevdepth = head.depth
return true
end
callback.register("post_linebreak_filter",
                  fix_nest_tail)
```

That is enough for our use case to work. For somebody interested in implementing a real replacement for the line breaking algorithm, additional adjustments are necessary; see the discussion in [28].

5 Conclusions

In 1990 the author attested that “the current” T_EX system is not powerful enough to meet all the challenges of high quality (hand) typesetting.

👉 Two decades later the successors offer significant improvements in that they provide machinery to resolve most of the issues identified.

👈 However, having the tools does not mean having the solutions and on the algorithmic level most questions are still unsolved.

So if they haven’t been solved for so long, are solutions truly needed?

In the author’s opinion, the answer is clearly *yes*. The fact that for nearly all issues people struggled again and again with inadequate ad hoc solutions shows that there is interest in better and easier ways to achieve the desired results. Or, to paraphrase Frank Zappa, “High-quality typography is not dead, it just smells funny”.

👉👉👉 The task now is to put the new possibilities to use and work on solving the open questions with their help.

* * *

The author wants to thank Nelson Beebe, Karl Berry, and Barbara Beeton for their invaluable help in improving the paper through thorough copy-editing and numerous suggestions.

References

- [1] Anonymous. $\epsilon\chi$ T_EX. Website. <http://www.extex.org>.
- [2] Anonymous. Extended T_EX font encoding scheme — Latin, Cork, September 12, 1990. *TUGboat*, 11(4):516–516, November 1990. <http://tug.org/TUGboat/tb11-4/tb30ferguson.pdf>.
- [3] Anonymous. Cambria (typeface). Wikipedia, 2012. [http://en.wikipedia.org/wiki/Cambria_\(typeface\)](http://en.wikipedia.org/wiki/Cambria_(typeface)).
- [4] Anonymous. LuaT_EX on the Web. Website, 2012. <http://luatex.org>.
- [5] Anonymous. Microtypography. Wikipedia, 2012. <http://en.wikipedia.org/wiki/Microtypography>.

- [6] Anonymous. X_YTeX on the Web. Website, 2012. <http://tug.org/xetex>.
- [7] Anne Brüggeman-Klein, Rolf Klein, and Stefan Wohlfeil. Pagination reconsidered. *Electronic Publishing*, 8(2&3):139–152, September 1995. <http://cajun.cs.nott.ac.uk/compsci/epo/papers/volume8/issue2/2point9.pdf>.
- [8] David Carlisle. A seasonal puzzle: XII. *TUGboat*, 19(4):348–348, December 1998. <http://tug.org/TUGboat/tb19-4/tb61carl.pdf>.
- [9] Paolo Ciancarini, Angelo Di Iorio, Luca Furini, and Fabio Vitali. High-quality pagination for publishing. *Software—Practice and Experience*, 42(6):733–751, June 2012.
- [10] Michael Downes and Morten Høgholm. *The breqn package*. CTAN, 2008. <http://www.ctan.org/pkg/breqn>.
- [11] Hisato Hamano. Vertical typesetting with TeX. *TUGboat*, 11(3):346–352, September 1990. <http://tug.org/TUGboat/tb11-3/tb29hamano.pdf>.
- [12] Charles Jacobs, Wilmot Li, Evan Schrier, David Barger, and David Salesin. Adaptive grid-based document layout. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH’03, pages 838–847, New York, NY, USA, 2003. ACM. <http://grail.cs.washington.edu/pub/papers/Jacobs2003.pdf>.
- [13] Donald Knuth. Virtual Fonts: More Fun for Grand Wizards. *TUGboat*, 11(1):13–23, April 1990. <http://tug.org/TUGboat/tb11-1/tb27knut.pdf>.
- [14] Donald Knuth. An earthshaking announcement. *TUGboat*, 31(2):121–124, 2010. <http://tug.org/TUGboat/tb31-2/tb98knut.pdf>. Also available as video at <http://river-valley.tv/tug-2010/an-earthshaking-announcement>.
- [15] Donald Knuth and Pierre MacKay. Mixing right-to-left texts with left-to-right texts. *TUGboat*, 8(1):14–25, April 1987. <http://tug.org/TUGboat/tb08-1/tb17knutmix.pdf>.
- [16] Donald E. Knuth. TAU EPSILON CHI. A system for technical text. Report STAN-CS-78-675, Stanford University, Department of Computer Science, Stanford, CA, USA, 1978.
- [17] Donald E. Knuth. *TeX and METAFONT—New Directions in Typesetting*. Digital Press, 12 Crosby Drive, Bedford, MA 01730, USA, 1979.
- [18] Donald E. Knuth. *The TeXbook*. Addison-Wesley, Reading, MA, USA, 1984.
- [19] Donald E. Knuth. *TeX: The Program*, volume B of *Computers and Typesetting*. Addison-Wesley, Reading, MA, USA, 1986.
- [20] Donald E. Knuth. The new versions of TeX and METAFONT. *TUGboat*, 10(3):325–328, November 1989. <http://tug.org/TUGboat/tb10-3/tb25knut.pdf>.
- [21] Donald E. Knuth. *Digital Typography*. CSLI Publications, Stanford, CA, USA, 1999.
- [22] Krista Lagus. Automated pagination of the generalized newspaper using simulated annealing. Master’s thesis, Helsinki University of Technology, Helsinki, Finland, 1995. <http://users.ics.aalto.fi/krista/personal/dippa/DITY0.ps.gz>.
- [23] Franklin Mark Liang. *Word Hy-phen-a-tion by Com-pu-ter*. Ph.D. dissertation, Computer Science Department, Stanford University, Stanford, CA, USA, March 1984. <http://tug.org/docs/liang>.
- [24] Mico Loretan. The selnolig package: Selective suppression of typographic ligatures. Website, 2012. <http://meta.tex.stackexchange.com/questions/2884>.
- [25] Kim Marriott, Peter Moulder, and Nathan Hurst. Automatic float placement in multi-column documents. In *Proceedings of the 2007 ACM symposium on Document engineering*, DocEng’07, pages 125–134, New York, NY, USA, 2007. ACM. <http://bowman.infotech.monash.edu.au/~pmoulder/examples/float-placement.pdf>.
- [26] Frank Mittelbach. E-TeX: Guidelines for future TeX extensions. *TUGboat*, 11(3):337–345, September 1990. <http://tug.org/TUGboat/tb11-3/tb29mitt.pdf>.
- [27] Frank Mittelbach. Formatting documents with floats: A new algorithm for L^ATeX 2_ε. *TUGboat*, 21(3):278–290, September 2000. <http://tug.org/TUGboat/tb21-3/tb68mittel.pdf>.
- [28] Frank Mittelbach. `\lastnodetype` not working as expected in LuaTeX. Website, 2012. <http://tex.stackexchange.com/questions/59176>.
- [29] Frank Mittelbach, Michel Goossens, Johannes Braams, David Carlisle, Chris Rowley, Christine Detig, and Joachim Schrod. *The L^ATeX Companion*. Tools and Techniques

for Computer Typesetting. Addison-Wesley, Reading, MA, USA, second edition, 2004.

- [30] Michael F. Plass. *Optimal pagination techniques for automatic typesetting systems*. Ph.D. dissertation, Computer Science Department, Stanford University, Stanford, CA, USA, 1981.
- [31] Jan Michael Rynning. Proposal to the TUG meeting at Stanford. *T_EXline*, 10:10–13, May 1990. Reprint of the paper that triggered T_EX 3.0.
- [32] Yasuki Saito. Report on jT_EX: A Japanese T_EX. *TUGboat*, 8(2):103–116, July 1987. <http://tug.org/TUGboat/tb08-2/tb18saito.pdf>.
- [33] Robert Schlicht. *Microtype; an interface to the micro-typographic extensions of pdfT_EX*, 2010. <http://ctan.org/pkg/microtype>.
- [34] Manfred Siemoneit. *Typographisches Gestalten*. Polygraph-Verlag, Frankfurt am Main, Germany, second edition, 1989.
- [35] Michael Spivak. `amstex.doc`, 1990. Comments to [36].
- [36] Michael Spivak. `amstex.tex`, 1990.
- [37] Hán Thế Thánh. Microtypographic extensions to the TeX typesetting system. (Dissertation an der Fakultät Informatik, Masaryk University, Brno, Oktober 2000). *TUGboat*, 21(4):317–434, 2000. <http://tug.org/TUGboat/tb21-4/tb69thanh.pdf>.
- [38] Hán Thế Thánh. Margin kerning and font expansion with pdfT_EX. *TUGboat*, 22(3):146–148, September 2001. <http://tug.org/TUGboat/tb22-3/tb72thanh.pdf>.
- [39] Hán Thế Thánh. Micro-typographic extensions of pdfT_EX in practice. *TUGboat*, 25(1):35–38, 2004. <http://tug.org/TUGboat/tb25-1/thanh.pdf>.
- [40] Arno Trautmann. An overview of T_EX, its children and their friends. Website, 2012. <http://github.com/alt/tex-overview>.
- [41] Stefan Wohlfeil. *On the Pagination of Complex, Book-Like Documents*. Shaker Verlag, Aachen and Maastricht, The Netherlands, 1998.
- [42] Hermann Zapf. About micro-typography and the hz-program. *Electronic Publishing*, 6(3):283–288, September 1993. <http://cajun.cs.nott.ac.uk/compsci/epo/papers/volume6/issue3/zapf.pdf>.

* * *

Finally, to answer the question posed in footnote 2: *The paragraph typeset with negative expansion was the second one in Section 3.4. Without it, it would have looked like this:*

The mark mechanism provides information about certain objects and their relative order on the current page, or more specifically, information about the first and last of these objects on the current page and about the last of these objects on any of the preceding pages. However, being a global concept only one class of objects can take advantage of the whole mechanism.

◇ Frank Mittelbach
Mainz, Germany
`frank.mittelbach` (at)
`latex-project` dot org
<http://www.latex-project.org/latex3.html>