TEX forever!

Jonathan Fine
Learning and Teaching Solutions
The Open University
Milton Keynes
United Kingdom
J.Fine@open.ac.uk
http://www.pytex.org

Abstract

This paper explores new ways of doing input to and output from TEX. These new ways bypass our current habits, and provide fresh opportunities.

Usually, TEX is run as a batch program. But when run as a daemon, TEX can be part of an interactive program. Daemons often that run forever, or at least for a long time. Hence the title of this paper.

Usually, parsing and transformation of the input data is done by TEX macros, with little outside help. Often, this results in input documents that only TEX can understand. Also, TEX macros can be hard to write. We demonstrate the replacement of TEX macros by an external program. This is done in real time.

Usually, TEX's principal output is a `dvi` representation of typeset pages, for processing by a printer driver. However, TEX's log file or console can be used to allow TEX to output the boxes it holds internally. (Alternatively, an extension of TEX could write this data out in a binary form.) Shipping out boxes rather than `dvi` allows an external program to do the page makeup.

Don Knuth's original conception was that TEX would be "just a typesetting language". In some sense he "put in many of TEX's programming features only after kicking and screaming". The developments described above reduce our dependence on TEX macros, and so bring our use of TEX closer to Knuth's original conception. Doing this will greatly improve its usefulness.

Long live TEX!

Introduction

In 1990, Don Knuth told us [8, p.572] that his work on developing TEX had come to an end. He went on to say:

> Of course I do not claim to have found the best solution to every problem. I simply claim that it is a great advantage to have a fixed point as a building block. Improved macro packages can be added on the input side; improved device drivers can be added on the output side.

In this paper, the author tries to follow this advice. There are imperfections in TEX, and the lack of proper support for Unicode fonts and filenames is a major weakness. However, TEX also has enormous strengths. It is archival. It carefully uses integer arithmetic to ensure that it gets the same line and page breaks, regardless of the machine it is running on. Its algorithm for breaking a paragraph into lines is reliable, adaptable and efficient. TEX is without rival for complex mathematical typesetting.

Often, TEX is used with LATEX as the macro package front end, and with `dvips` as the device driver. Sometimes, the word 'TEX' is used to refer to the whole system. However, in this paper we mean by 'TEX' the typesetting program written by Don Knuth. And so LATEX and `dvips` are tools for use with TEX.

This paper is concerned with making improvments on the input and output sides of TEX, both areas of work where there is an enormous amount still to do. However, our proposals are not exactly macro packages and device drivers.

A note to the reader: This paper has been written for a general audience, and in particular for those who are not TEX experts. At the same time, discussion of technical details is at times either unavoid-

1	¶	textfile → marked up text and math
2	†	text + transform → horizontal primitives
3	*	horizontal primitives → hlist
4	†	math + transform → math primitives
5	*	math primitives + parameters → hlist
6	*	hlist + parameters → vlist
7	†	vlists + page make up → page boxes
8	*	page boxes → sequential `dvi` file
9	¶	sequential `dvi` file → random access `dvi` file
10	¶	random access `dvi` file → rendered page

Table 1: How TEX works, in 10 stages
† usually done using TEX macros.
* usually done using TEX's built in procedures.
¶ file input and output matters.

able or helpful. Therefore, I hope that the experts will forgive my stating the obvious, and that the others forgive my discussing the difficult.

References: Many of the articles cited here have been reprinted in the collection *Digital Typography* [13]. Page numbers in citations refer to [13], and not to the original publication.

How TEX works

Table 1 gives a concise description as to how TEX works. On the input side we propose that an external program perform at least part of the transformation in steps 2 and 4.

On the output side we have two proposals. The first is that (8) be replaced by:

8′. page boxes → stream of `dvi` pages

The second, which is more ambitious, is that (7) be replaced by:

7′. vlist → external program

followed by page makeup in that external program.

Thus we continue to use TEX's excellent typesetting, but reduce the use of its macros.

TEX — just a typesetting language

TEX is a typesetting program, written by Don Knuth, that is particular good at mathematical and technical typesetting. TEX is reliable and stable, and is very widely used by academic mathematicians and physicists.

TEX has a macro programming language, which allows features to be added. The best known and most widely used TEX macro package is LATEX. (This is not quite accurate. Although originally LATEX used TEX, since 2003 it by default uses e-TEX, which is an extension of TEX. So it is no longer purely a TEX macro package. This has no bearing on our discussion.)

In 1996 Don Knuth, describing his intentions when he started to develop TEX, said [11, p.648]:

> I'm not going to design a programming language; I want to have just a typesetting language.

and at the same time he said (loc. cit.):

> In some sense I put in many of TEX's programming features only after kicking and screaming. [...] In the 70s, I had a negative reaction to software that tries to be all things to all people. Every system had its own universal Turing machine built into it somehow, and everybody's machine was a little different from everybody else's.

But the need for more features caused the programming constructs to grow (see Table 2 below). See also [16] for a 'wish-list' of future developments.

Therefore, by *removing* commands from TEX, we can come closer to Don's original conception of TEX. However, for this to succeed in practice, some other means of adding new features is required. Indeed, one of the major problems TEX users have now is that the existing programming constructs barely support the demand for new features. This we discuss later.

In this section we outline how to cut TEX down to the bare minimum. To be specific, in this section we ask: What commands are required in order to access TEX's algorithm for breaking a paragraph into lines?

To create a paragraph one needs to be able to load fonts, change fonts, and set a character in the current font. One also needs commands for appending glue, kerns and the like to the paragraph.

To break the paragraph into lines, one needs the `\par` primitive (also known as `\endgraf`) and a means of assigning values to the line-breaking parameters, such as `\hsize`.

In other words, the basic operations are to add an item to the current horizontal list, and to form a paragraph out of the current horizontal list. (For mathematics and table typesetting there are similar basic operations.)

It should at this point be clear that certain primitive TEX commands are *not* required in order to do typesetting. These commands include all the `\def` commands (such as `\def`, `\chardef`, `\xdef`), `\let`, `\begingroup` and `\endgroup`. Once category codes have been set up, there is no further need for `\catcode`. And there is certainly no need for commands such as `\expandafter`, `\noexpand`, `\aftergroup` and `\futurelet`. All these are *not*

Control sequence	Date added
\if	21 Jun 1978
\pausing	16 Mar 1978
\uppercase	25 Nov 1978
\xdef	28 Nov 1978
\ifmmode	23 July 1978
active characters	25 Jan 1980
\let	25 Mar 1980
\ifx	13 July 1981
\catcode	July 1982
\expandafter, \openin	12 Sep 1982
\string	12 Sep 1982
\immediate	12 Oct 1982
\csname, \endcsname, \fi	13 Nov 1982
\everymath, \everydisplay, \futurelet	2 Dec 1982
\endinput	7 Dec 1982
\jobname	25 Dec 1982
\globaldefs	20 Jan 1983
\iffalse, \iftrue	3 Feb 1983
\everyvbox, \everyhbox	6 Mar 1983
\everyjob	18 Mar 1983
\advance, \multiply, \divide \noexpand, \meaning	25 May 1983
\afterassignment	27 May 1983
\escapechar, \endlinechar	4 Jul 1983
\errhelp	11 Jul 1983
\aftergroup, \newlinechar	16 Jul 1983
\ifhbox, \ifvbox	27 Aug 1983
\holdinginserts	30 Sep 1989

Table 2: Some TEX control sequences not needed for typesetting (after [7])

typesetting commands, and exist only to allow features to be added to TEX.

Moreover, these commands cause difficulty for both TEX users and programmers. Their introduction is perhaps a sign that things were starting to go in the wrong direction.

In [7] Knuth published, in edited form, the log books he kept while he was developing TEX. In these, we can see the introduction of features. (See Table 2.)

Suppose all programming commands are disabled by \let-ting them be undefined, like so:

```
\let \afterassignment = \undefined
```

Provided we remove enough commands, we will have, as Don wanted TEX to be in the first place, "just a typesetting language". A language without features, and without the capability of adding features (which is itself a feature).

Comparison with PostScript and with machine code is instructive. Most PostScript is generated by programs that translate from a higher-level language down to PostScript. Similarly, much machine code is generated by compiling 'C' source files.

Many of us write input files for (LA)TEX, using a text editor. We won't do that for a featureless TEX. It's too much hard work, and anyway we want to write in a higher-level language. We are suggesting that an external program perform at least part of the text transformation that is traditionally performed by a TEX macro package.

Improved macros — input transformation

This section could also be titled:

```
\let \def = \undefined
```

Don suggested that we add improved macro packages on the input side. Now, a macro package has two main purposes. One is issuing typesetting instructions to TEX. This will create a galley (or page of unlimited depth). The second purpose is the output (or page makeup) routine, which breaks the galley into pages of a suitable size.

In this section we consider the creation of a galley. Marked-up text, such as

```
\section{Improved macros}
```

is translated (by LATEX in this case) into a large number of low-level instructions. The title

```
Improved macros
```

is scarcely translated. Each character sets itself, and space characters produce default interword glue. (Later, we present an example of this.)

It is \section{} that does most of the work. Here are some of the technical details. It selects the font to be used, and the paragraph parameters for the title (in case it is wider than the measure). It also places glue and penalties before and after the title on the galley. It might also add a section number, and record information for the table of contents.

High-level commands are being translated into low-level typesetting instructions. This translation need not be done by LATEX (or indeed by any other TEX macro package). For example, in the WEB system of literate programming, much of the work is done using external programs. Similarly, XSLT templates are often used to transform text, prior to it being passed to TEX to typesetting.

For over 10 years the LATEX3 project has been working to enhance LATEX by providing [15, p.1]

> a flexible interface for typographic designers to easily specify the formatting of a class of documents.

Such an interface might, for example, be similar to Cascading Style Sheets (CSS) for HTML. We have seen that Don Knuth only reluctantly added

programming features to TEX. The author believes that TEX macros are not a suitable language for creating the above interface, and that the long delay in its delivery is evidence for this.

This interface could instead be written as an external program. In a later section we describe QATEX, which is a wrapper around TEX that effectively allows TEX to interact with external programs.

Improved macros — output routines

This section could also be titled:

```
\let \output = \undefined
```

One of the most interesting and best parts of TEX is the algorithm it uses for breaking a paragraph into lines. The algorithm for breaking the galley into pages is not so good, although for simple technical material it is more than adequate.

In this paper we do not suggest improved output algorithms (we have discussed this elsewhere [4]). Instead, we describe a solution to a related problem. The TEX macro language is not a suitable environment for the writing of complicated page makeup algorithms. Here we describe a means of moving the problem to another domain.

The galley produced by TEX consists of a vertical list. This vertical list consists of boxes, glue, penalties, and so forth (see *The TEXbook*, page 110 for a complete list). The `\showlists` command prints a detailed description of the content of this vertical list.

The output of `\showlists` can parsed by an external program, and used to reconstruct within that program the vertical list created by TEX. If the external program can also send low-level typesetting instructions to TEX, then TEX in effect has become a callable function available to the external program. (As in QATEX, the interactive console or more exactly stdin and stdout can be used for this communication.)

This is not a completely new idea. In 1996, Jiří Veselý asked [10, pp620–621]:

> Once I was asked about the possibility to make a list of all hyphenated words in the book. I was not able to find a way in your book to do this.

To this, Don replied (loc. cit.):

> This would be easy to do now in a module specially written for TEX. I would say that right now, in fact, you could get almost exactly what you want by writing a filter that says to TEX "Turn on all the tracing options that cause it to list the page contents." Then a little filter program would take the trace information through a UNIX pipe and it would give you the hypenated words. It would take an afternoon to write this program; well, maybe two afternoons . . . and a morning.

We develop this idea later in the paper.

Instant Preview and TEX as a daemon

This section could also be titled:

```
\let \end = \undefined
```

Interactive programs typically require a response time of less than a tenth of a second, while a response in a hundreth of a second is seen as instantaneous.

On my current 800 MHz PC, the command

```
$ tex story \\end
```

takes about 0.137 seconds, while

```
$ tex \\end
```

takes 0.133 seconds. The first command typesets a small page of material; the second does nothing but start TEX and then exit. Thus, typesetting the small page takes about 0.004 seconds.

It follows from this that typesetting material for Instant Preview is tolerable if the start-up time is included, while it will be perceived as instantaneous if TEX is run as a daemon.

Running TEX as a daemon is an example of TEX forever. We wish for TEX to start up when the computer boots, and to remain running indefinitely. Moreover, we might prefer that there were only a single instance of the TEX daemon running.

Documents and macro packages may have to be adapted, to make the most of Instant Preview. The key concept seem to be this: That the source file be partitioned into regions by markers, which we call 'belays', and that the macro package be able to typeset each region independently. In other words, that the macro package support random access typesetting. This is, again, an example of TEX being enhanced by an improved macro package on the front end.

The author has already written [5] about Instant Preview. At the conference he hopes to demonstrate the latest progress.

Decorating `dvi` files

TEX has no built-in notion of colour, or of graphics inclusion. However, the `\special` command allows device drivers to produce special effects. By decoration we mean the application of colour, change bars and the like to the rendered page.

In the domestic setting, decoration of a room or a house does not move the walls or make other structural alterations. In typesetting, adding decoration should not affect typesetting decisions, such as the line breaks and the placement of items on the page. (This is not to say that the typographic design should not take into account the subsequent application of decoration.)

The current practice regarding decoration is to use a fairly simple `dvi` processor, and to have the (LA)TEX macro package place appropriate `\special` commands into the `dvi` file. From the point of view of a device-driving `dvi` processor, this is probably correct. It seems that, historically, low-level capabilities were added to device drivers. Then macro packages were written to access these new features.

From the point of view of the macro package, this approach is probably wrong. As already noted, decoration should not affect typesetting decisions. This is an important property, whose fulfilment should be central to the approach taken.

Suppose, for example, that some text is to be printed in a spot colour. Placing a `\special` at the start or end of a word does not affect its hyphenation. Therefore something like

```
% usage: \color{red}{Text to go in red}
\def\color#1#2{%
    \special{push #1}%
    #2%
    \special{pop}%
}
```

will suffice, at least in the simplest cases.

However, a page boundary might break the red text. This places a burden on the `dvi` processor, to keep track of this information. Typical `dvi` processors allow random access to the pages in the `dvi` file. Having to look at previous page(s) breaks this random access.

The solution we suggest is to write a `dvi`-to-`dvi` filter that resolves these random access problems. Such a filter is not, of course, a device driver, but it can be used with any device driver. Its purpose is to translate high-level specials into low-level specials.

TEX is being held back by the weakness of tools for decorating text. For example, a common requirement is to place a background rectangle behind a paragraph of text. If the paragraph is broken over two pages, the background rectangle should be similarly broken.

In 1987 Don Knuth and Pierre MacKay discussed a similar problem, namely implementing bi-directional typesetting without extending TEX. They wrote [14, p159]:

> How can we get TEX to do this? The best approach is probably to extend the driver programs that produce printed output from the `dvi` files that TEX writes, instead of trying to do tricky things with TEX macros.

In the same article [13, p161] they then produced a "much more reliable and robust scheme by building a specially extended version of TEX".

QATEX — or ask a friend a question

TEX is a typesetting language, with limited text-processing and other capabilities. Things that are easy to do in other languages are hard to do in TEX. Examples are to find the dimensions of an EPS or other graphics file, or to calculate the sine and cosine of an angle, so that space can be left for rotated text.

Traditionally, such questions have been answered by writing TEX macros. The author finds that TEX macros are not a suitable language for such text manipulation tasks.

Here is an extract from the `\Gread@eps` macro in the LATEX file `graphics.sty`.

```
\immediate\openin\@inputcheck#1 %
\ifeof\@inputcheck
  \@latex@error{File `#1' not found}%
     \@ehc
\else
   \Gread@true
   \let\@tempb\Gread@false
   \loop
     \read\@inputcheck to\@tempa
     \ifeof\@inputcheck
       \Gread@false
     \else
       \expandafter\Gread@find@bb
         \@tempa:.\\%
     \fi
   \ifGread@
   \repeat
   \immediate\closein\@inputcheck
\fi
```

The author has developed QATEX, which allows the TEX macro programmer to ask another process to answer questions. Such as: What is the bounding box of `myfile.eps`?

QATEX (pronounced 'kwa-tech') provides a different route. Questions and answers are the essence of QATEX. When QATEX is used, lines such as:

```
!Q=qatexlib.eps.bbox,myfile.eps
!A=0,0,0 0 35 97
```

appear in the TEX's `log` file.

The first line is a question, produced using a `\write` command. The second line is the answer.

The characters `!A=` are a prompt, produced using a `\message` command.

The remainder of that line is the answer to the question. The prefix `0,0,` tells TEX that the question was successfully posed and answered. There follows the information asked for. This information is supplied by a process external to TEX.

TEX uses `\read -1 to \temp` to read this information from its stdin stream. So far as TEX is concerned, this data might have come from the user. In fact, it has come from a program, namely QATEX.

QATEX works as follows. It is a wrapper progam that invokes TEX, and takes control of its standard input and output. When it sees a question line, followed by the answer prompt, it parses and answers the question, then sends the answer to TEX, using TEX's standard input. However, QATEX does not answer the question itself. It imports a module — in the example above the `eps` module — to answer the question for it.

Here is the definition, in Python, of a function that returns the bounding box of an EPS file, as a string. If not found, it raises an exception. It took me less than 10 minutes to write. It would be part of a `eps` module for use with QATEX.

```
# File: qatexlib/eps.py
_bb_prefix = '%%BoundingBox: '
def qa_bbox(filename):
  f = open(filename)
  for line in f:
    if line.startswith(_bb_prefix):
      sizes = line.split()[1:]
      return ' '.join(sizes)
  msg = "File '%s' has no bounding box"
  raise Exception, msg % filename
```

Alternatives and complements to QATEX

In this section we compare QATEX to shell escape, `eval4tex`, PerlTEX, `sTeXme` and Pymacs. Each of these has some similarity with QATEX, and informed the design and implementation of QATEX.

Shell escape. Modern implementations allow TEX to issue shell commands, as if they had been typed at a command prompt. This allows, for example, a command such as `makeindex` to be run after TEX has processed the body of a document, but before setting the back matter.

However, it also allows other commands to be run, such as the deletion of files. And TEX documents can execute arbitary TEX commands. (Strictly speaking, this is not true. For example, in Active TEX [3] all characters are active. This allows a macro package to prevent execution directly from a document of all but specified commands.)

Often, TEX documents are distributed in source form. If shell escape is enabled, the typesetting a document could result is a shell escape command being run, that finds and deletes all your files. Clearly, shell escape is a security risk. For this reason, shell escape is disabled by default, and is rarely used.

Even without shell escape, TEX macros and therefore documents can overwrite existing files. (TEX has no inbuilt ability to delete files. But it can destroy their contents.) Therefore, modern implementations of TEX refuse to open for writing files that are not in or beneath the current directory, or a similarly specified area.

This restriction is not applied to the reading of files. Therefore, it is possible for a TEX document, when typeset, to include in it other files. These other files might be confidential.

Therefore, in line with the theme of TEX being "just a typesetting system", it might be sensible to:

```
\let \openout = \undefined
\let \openin = \undefined
\let \input = \undefined
```

and have another program take responsibility for security. The security monitor could then send material to be typeset to TEX's standard input stream.

eval4TEX (by Dorai Sitaram) is a two-pass process that allows TEX to send expressions to Scheme for evaluation [1]. It provides a `\eval` macro, that is used as below. (The example is Sitaram's, and my exposition follows his).

```
\eval{(display (acos -1))}  % gives pi
```

On the first pass, the Scheme code

```
(display (acos -1))
```

is written to an auxiliary file, together with some indexing information.

Before the second pass, a helper program `eval4tex` runs Scheme on the auxiliary file, to create a second auxiliary file. On the second pass, the `\eval` command picks up values from the second auxiliary file, and refreshes the data in the first.

As Sitaram writes:

> This strategy is quite common in the TEX world. The popular TEX-support programs BibTeX and MakeIndex, which generate bibliographies and indices respectively, both operate this way.

sTeXme (by Oleg Paraschenko) is another approach to integrating TEX with Scheme [19]. Here is his summary of the goals of the project.

> The (LA)TEX macro language was a great development when it appeared, but now it is too out-of-date. Programming in TEX is a fun, but more often it is a pain.
>
> As it seems for me, only very few people can write (LA)TEX macros, but a lot of people would like doing it (like me, for example). This is the problem.
>
> One of the solutions is to provide another scripting language for TEX. That's what is the goal of the sTeXme project. It should provide the Scheme programming language as a TEX scripting language.

This project has two parts, namely an extension to TEX, that allows it to interpret Scheme code, and an extension to Scheme that allow it access TEX internals. We say more on this later.

PerlTEX (by Scott Pakin) uses standard Perl and TEX without extensions [17]. Here is his summary of the goals of the project.

> TEX is a professional-quality typesetting system. However, its programming language is rather hard to use for anything but the most simple forms of text substitution. Even LATEX, the most popular macro package for TEX, does little to simplify TEX programming.
>
> Perl is a general-purpose programming language whose forte is in text manipulation. However, it has no support whatsoever for typesetting.
>
> PerlTEX's goal is to bridge these two worlds. It enables the construction of documents that are primarily LATEX-based but contain a modicum of Perl. PerlTEX seamlessly integrates Perl code into a LATEX document, enabling the user to define macros whose bodies consist of Perl code instead of TEX and LATEX code.

Here is Scott Pakin's equivalent to `\eval`:

```
\perlnewcommand{\reversewords}[1]
  {join " ", reverse split " ", $_[0]}
\reversewords{TeX forever!}
```

PerlTEX, like QATEX, invokes TEX under the control of a separate process. Unlike QATEX, it does not take control of TEX's standard input and output. Invoking `\reversewords` causes TEX to write material to a specially named file. This file corresponds to the question in QATEX. The controlling Perl process then computes the answer to the question, and writes it to another specially named file. Meanwhile, the TEX process goes into a loop, to poll for the existence of the answer file. Once found, it is `\input` by TEX.

PerlTEX seems to have a performance problem. On my 800 Mhz PC, the following example:

```
\documentclass{article}
\usepackage{perltex}
\perlnewcommand{\nothing}{}

\begin{document}
% I've got plenty of nothing ...
\nothing\nothing\nothing
\nothing\nothing\nothing
\nothing\nothing\nothing
\nothing           % 10 nothings
% We're busy doing nothing ...
\end{document}
```

takes about 3.0 seconds to execute. This includes startup time. (On the same machine, it takes QATEX about 1/2500 seconds to do nothing once.)

Here is at least a partial explanation. Instrumenting the code for PerlTEX shows that in compiling the above document, TEX polls for the existence of the helper file approximately 5,000 times. The exact number varies. Adding:

```
\input nothing  % input an empty file
```

to the polling loop *reduces* the time taken to about 1.8 seconds, and reduces the number of pollings to about 500. The UNIX `nice` command could also help here.

Pymacs (by François Pinard) is not a way of using Python with TEX. It is a way of using Python with the Emacs editor [18]. To quote its author:

> Pymacs is a powerful tool which, once started from Emacs, allows both-way communication between Emacs Lisp and Python. Pymacs aims Python as an extension language for Emacs rather than the other way around, and this asymmetry is reflected in some design choices. Within Emacs Lisp code, one may load and use Python modules. Python functions may themselves use Emacs services, and handle Emacs Lisp objects kept in Emacs Lisp space.

Pymacs is mentioned because is was higly influential on the author's approach to the integration of TEX with a scripting language. (At that time, Python had not been chosen.)

Different approaches compared

In the previous two sections we looked at QATEX, shell escape, `eval4tex`, `sTeXme` and PerlTEX. In this section we make some comparisions.

Philosophy QaTeX is like shell escape, in that simple queries are sent to another process. The other approaches assume that complex code will be written within, or otherwise produced by, TeX macros. This code is then evaluated by another program.

For example, in QaTeX the problem of reversing words might result in the following conversation between TeX and the external process:

```
!Q=qatexlib.string.reverse,TeX forever!
!A=0,0,forever! TeX
```

The question sent to QaTeX could not be along the lines of: "What is the result of evaluating this complex Perl or Scheme expression?" However, such is not the expected use. Rather, it is expected that TeX will send a short and simple query. If the answer is long, it could be placed in an external file. Once that is done, TeX can be told, as the answer, that the file is ready to be `\input` (assuming `\input` is still defined).

Architecture The architecture of the implementation of PerlTeX is closer to that of `eval4tex` than that of `sTeXme`. Perl code is placed in the body of TeX macros, and this code is sent out to Perl for evaluation. Unlike `sTeXme`, and like `eval4tex`, it does not require either an extension to TeX or a special version of the command interpreter for the extension language.

PerlTeX is similar to QaTeX in that TeX is run under the control of an external program. However, QaTeX uses standard input and output for communication, whereas PerlTeX polls named files.

QaTeX provides an efficient and portable low-level interface between TeX and an external process. PerlTeX is a higher-level package. There is no reason why the QaTeX interface, or something similar to it, should not be used by PerlTeX, so as to improve performance. The same applies to `eval4tex`, where using this interface would remove the need for a second run. It would also provide better interaction.

Security Any process that evaluates code supplied by a document will expose a security problem, unless the code evaluator is already secure (as in Java, for example). PerlTeX provides security by using a secure Perl sandbox. QaTeX provides security by having TeX (and thus the document) supply only data.

Of course, any program that evaluates untrusted data as if it were trusted code has a security issue. If it is necessary to evaluate safely untrusted code, then a secure sandbox is required. This is the key security issue. QaTeX is a small interface application, which does not attempt to solve this problem. There is no reason why QaTeX should not be used with such a secure sandbox. But that is a matter for the developer who builds upon QaTeX.

Integration and extension Of all the projects considered in this section, `sTeXme` is the most ambitious. It involves making major extensions to TeX, to produce a new program, called `sTeXme`.

The new name is necessary. TeX experts will already know that Don Knuth does not object to the sources of TeX the program being used as the basis for creating a superior program. However, he is most insistent that programs that are not TeX must not be called TeX. More exactly, in [8, p572] he wrote:

> That is all I ask, after devoting a substantial portion of my life to the creation of these systems and making them available to everybody in the world. I sincerely hope that the members of TUG will help me to enforce these wishes, by putting severe pressure on any person or group who produces any incompatible system and calls it TeX or METAFONT or Computer Modern — no matter how slight the incompatibility might seem.

This insistence on the stability and consistency of TeX is, in this author's view, a significant contribution to its longevity. Users know what to expect, and get what they expect, when they use TeX.

Regarding the scope of his new program `sTeXme`, Oleg Paraschenko reports:

> [...] Scheme code can be executed from TeX and that Scheme code can access TeX internals (getting a string from the TeX string pool, getting a macro definition as the Scheme list).

The source file `stexmelib.c` on the SourceForge repository indicates that Paraschenko is building a C-coded Scheme extension, in which equivalents to TeX boxes and the like can be stored. This indicates that there are many points of contact between his project and the next section.

PyTeX

The goal of the PyTeX project is to combine Python programming with TeX typesetting. To understand this, think of Tcl/Tk: Tcl is a scripting language and Tk is a toolkit for building GUI programs. Perl and Python also have interfaces to Tk, allowing them to use Tk when building GUI programs.

Now think of LaTeX as LA/TeX. LA is a front end to the TeX typesetting program written in TeX's

macro language. PyTEX, or Py/TEX if you prefer, is to be a front end to TEX written in Python.

PyTEX replaces the part of TEX that Don Knuth said he did not want to write, namely the TEX macro programming language, with something more widely used. Our aim is to provide an alternative means of programming typesetting decisions and logic. This will make TEX easier to use.

Here is an example of the interface. We are in Python, and wish to typeset a paragraph of text, namely

The cat sat on the mat.

to a measure of 6 picas, which is about one inch (or 25 millimetres). This is a toy example. After typesetting the paragraph, we wish to bring it into Python for page makeup.

To see how this is done, issue the command

```
$ cat cat_sat.tex | tex | grep '^[.\]'
```

where the source stream is

```
% cat_sat.tex
\tracingonline 1
\showboxbreadth \maxdimen
\showboxdepth \maxdimen
\scrollmode

\setbox0\vbox{%
\hsize  6pc
The cat sat on the mat.\par
\showlists
}
```

The annotated output of the `grep` is:

```
# This is an annotation.
# Start of the first line of paragraph.
\hbox(6.94444+0.0)x72.0, glue set 0.58331
# indentation box, 20pt wide
.\hbox(0.0+0.0)x20.0
# The word "The".
.\tenrm T
.\tenrm h
.\tenrm e
# normal interword glue
.\glue 3.33333 plus 1.66666 minus 1.11111
# The word "cat".
.\tenrm c
.\tenrm a
.\tenrm t
.\glue 3.33333 plus 1.66666 minus 1.11111
.\tenrm s
.\tenrm a
.\tenrm t
# Zero width line filling glue.
.\glue(\rightskip) 0.0
# Penalty for breaking page at this point.
\penalty 300
# Interline glue.
\glue(\baselineskip) 5.05556
# Start of second line of paragraph.
\hbox(6.94444+0.0)x72.0, glue set 20.88878fil
.\tenrm o
.\tenrm n
.\glue 3.33333 plus 1.66666 minus 1.11111
.\tenrm t
.\tenrm h
.\tenrm e
.\glue 3.33333 plus 1.66666 minus 1.11111
.\tenrm m
.\tenrm a
.\tenrm t
.\tenrm .
# Inserted by TeX, for internal reasons.
.\penalty 10000
# Allow the last line of para to be short.
.\glue(\parfillskip) 0.0 plus 1.0fil
.\glue(\rightskip) 0.0
```

This is a complete description of the paragraph formed by TEX's line breaking algorithm.

This is the essence of the interface between Python and TEX. Material is sent to TEX to be typeset, say using stdin. The `\showlists` command is used to write the results to stdout, from which they are picked up by Python.

On the input side, Python is responsible for parsing the input stream, and placing appropriate items on the horizontal list. It is also responsible for ensuring that nothing inappropriate is placed on the list. The whole process should not generate TEX errors (although warnings about overfull boxes and so forth are welcome).

On the output side, Python parses the output stream, and from it reconstitutes the boxes that TEX has formed, thus forming a Python object.

In Python code, our example might look like

```
hlist = Hlist()  % new horizontal list
text = "The cat sat on the mat."
hlist.extend(text)
vlist = hlist.linebreak(hsize=6*pica)
```

where hidden in the call to the `linebreak()` method there is a sending of data to TEX, and a reconstruction in Python of the boxes that TEX built. From here on, the output or page-makeup routine can be written in Python. Note that `cat_sat.tex` invokes no TEX macros.

Conclusions

In the 15 years since TEX has been frozen, very few deficiencies have been found in the algorithms it uses for breaking a paragraph into lines, for typesetting mathematics, and for setting tables. Since 1990 there has been little (but valuable) progress in

the area of Unicode fonts, for which an extension of TEX genuinely is needed. TEX was written to be archival, and it is holding up well after its first quarter-century or so.

There are many problems in our use of TEX. This paper has discussed several:

- coloured text and other decorations,
- interactive use of TEX,
- input transformation,
- programming page makeup.

All of these arise not out of TEX itself, but out of the way in which we use TEX.

There is an irony in our use of TEX macros. Recall that when Don was looking at the design of TEX he found that [11, p.648]:

> Every system I looked at had its own universal Turing machine built into it somehow, and everybody's machine was a little different from everybody else's.

He then went on to say:

> I was tired of having to learn yet another almost-the-same programming language for every system I looked at; I was going to try to avoid that.

What we have now with TEX macros is a Turing machine very different from any other. This is just what he wished to avoid. However, QATEX provides a powerful complement to existing TEX macro packages, and PyTEX will use TEX as "just a typesetting language", which is what Don wanted it to be in the first place.

In 1996, Piet van Oostrum asked Don Knuth about TEX's macro programming language [11, p648–9]:

> I don't know if you have ever looked into the LATEX code inside, but if you look into that, you get the impression that TEX is not the most appropriate programming language to design such a large system. Did you ever think of TEX being used to program such large systems and if not, would you think of giving it a better programming language?

In response to this, Don Knuth said (loc. cit.):

> It would be nice if there were a well-understood standard for an interpretive programming language inside an arbitary application. Take regular expressions — I define UNIX as "30 definitions of regular expressions living under one roof." [*laughter*] Every part of UNIX has a slightly different regular expression. Now, if there were a universal simple interpretive language that was common to other systems, naturally I would have latched onto that right away.

The theme of this paper is TEX typesetting with fewer macros. We use instead a "simple interpretive language", namely Python. If we learn to use TEX in new ways, and take good care of it, TEX will be good for its second quarter-century.

Long live TEX!

References

[1] `eval4tex`, `http://www.ccs.neu.edu/home/dorai/tex2page/eval4tex-doc.html`

[2] Jonathan Fine, Editing `.dvi` files, or Visual TEX, *TUGboat*, **17** (1996), 255–259

[3] ———, Active TEX and the DOT input syntax, *TUGboat*, **20** (1999), 248–254

[4] ———, Line breaking and page breaking, *TUGboat*, **21** (2000), 210–221

[5] ———, Instant Preview and the TEX daemon, *EuroTEX 2001 Conference Proceedings*, 49–58

[6] ———, TEX as a callable function, *EuroTEX 2002 Conference Proceedings*, 26–35

[7] Donald E. Knuth, The Errors of TEX, *Software — Practice and Experience*, **19** (1989), 607–685. (Reprinted in [9])

[8] ———, The Future of TEX and METAFONT, *TUGboat*, **11** (1990), 489 (Reprinted in [13])

[9] ———, *Literate Programming*, CSLI (1992)

[10] ———, Questions and Answers II, *TUGboat*, bf 17 (1996), 355-367 (Reprinted in [13])

[11] ———, Questions and Answers III, *MAPS (Minutes and APpendiceS)*, **16** 1996, 38–49 (Reprinted in [13])

[12] ———, The future of TEX and METAFONT, *TUGboat*, **11** (1990), 489 (reprinted in [13])

[13] ———, *Digital Typography*, CSLI (1999)

[14] Donald E. Knuth & Pierre MacKay, Mixing Right-to-Left Texts with Left-to-Right Texts, *TUGboat*, **8**, (1987), 14–25. (Reprinted in [13])

[15] Frank Mittelbach & Chris Rowley, The LATEX3 Project, `http://www.latex-project.org/guides/ltx3info.pdf`

[16] NTG TEX future working group, TEX in 2003: Part I Propositions and Conjectures on the Future of TEX, *MAPS (Minutes and APpendiceS)*, **21** 1998, 13–19

[17] PerlTEX, `http://www.ctan.org/tex-archive/macros/latex/contrib/perltex`

[18] Pymacs, `http://pymacs.progiciels-bpi.ca`

[19] sTeXme, `http://stexme.sourceforge.net`