25 Years of TEX and METAFONT: Looking Back and Looking Forward: TUG'2003 Keynote Address

Nelson H. F. Beebe
University of Utah
Department of Mathematics, 110 LCB
155 S 1400 E RM 233
Salt Lake City, UT 84112-0090
USA
WWW URL: `http://www.math.utah.edu/~beebe`
Telephone: +1 801 581 5254
FAX: +1 801 581 4148
Internet: `beebe@math.utah.edu, beebe@acm.org, beebe@computer.org`

Abstract

TEX has lasted longer than many other computer software technologies.

This article reviews some of the history of TEX and METAFONT, how they have come to be used in practice, and what their impact has been on document markup, the Internet, and publishing.

TEX has several design deficiencies that limit its use and its audience. We look at what TEX did right, and with 25 years of hindsight, what it did wrong.

We close with some observations about the challenges ahead for electronic representation of documents.

Contents

HOME OF "ONO KINE" FOOD AND MALASADAS
TEX
Drive In
HIGHWAY 19 HONOKAA BIG ISLAND OF HAWAII

Figure 1: The TEX drive-in has two locations on the Big Island, one in Honokaa in the northeast, and one in Pahala, near the south center. It is noted for malasadas, a puffy hole-less donut, brought to Hawaii by Portuguese agricultural workers. One Web site reports that "Tex sells more malasadas than some McDonald's outlets sell hamburgers."

Foreword

Some of the material in this article may seem old hat to veteran TeX users, but I am writing it with the intention that it can also be read by people who are unfamiliar with TeX and METAFONT, but are nevertheless interested in learning something about the design and history of those programs.

Introduction

The TUG'2003 Conference at the Waikoloa Beach Marriott Hotel on the northwest coast of Hawaii (the Big Island) celebrated the 25th anniversary of TeX and METAFONT, and the 24th anniversary of the TeX Users Group. It was amusing to discover that TeX already had a commercial presence there: see Figure 1.

Donald Knuth enjoys finding numerical patterns, so I looked for some in connection with this meeting. The year 2003 contains the first two primes, and two zeros. Two is the base of most computer number systems, and 2003 is also the first prime in this millenium. In base 2, $2003 = 11\,111\,010\,011_2$ and $25 = 11\,001_2$: their five low-order bits are mirror images of each other. The number 25 is 5^2, or $(\text{third prime})^{(\text{oddest prime of all})}$.

Some historical highlights

Document production by humans goes back a rather long way, as shown in Tables 1 and 2. Although paper was invented about 6000 years ago, it was not until the middle of the 19th Century that wood pulp became the primary source of paper, and it took a few decades longer for paper to become widely available at low cost.

Gutenberg's invention predated Columbus' discovery of the Western World by just 40 years, and made large-scale book production practical. Before Gutenberg, each book was copied by hand; after Gutenberg, literacy was no longer restricted to a privileged class, and societal progress was poised for a huge leap forward.

It took about 30 years after the invention of digital computers for the first effective document formatting and typesetting systems to be developed. TeX and METAFONT were first implemented during Donald Knuth's 1977–78 sabbatical year. They were written in the SAIL[1] programming language, which was available only on DEC-10 and DEC-20 computers with PDP-10 CPUs. I had the good fortune in

[1] (Stanford Artificial Intelligence Lab/Language)

Table 1: Notable historical events BC (before computers).

Year	Event
4000 BCE	Egyptians invent papyrus from woven reeds
105	Ts'ai Lun invents bark/hemp/rags-based paper in China
1009	First European paper mill, in Xativa, Spain
1411	First paper mill in Germany
1452	Johannes Gutenberg invents movable type
1680	First paper mill in New World, in Culhuacan, Mexico
1690	First paper mill in English colonies, near Philadelphia
1798	Nicholas Robert invents first paper-making machine, in France
1850–1879	Paper from wood pulp perfected
1889–1900	Economical mass-produced paper

1978 or 1979 to hear a talk that he gave at Xerox PARC about TEX, and I realized immediately that older document production systems, like IBM's ATS, DEC's `runoff`, and my own `document`, would be obsolete as soon as TEX were widely available. We had a DEC-20 at Utah, so we had early access to the Stanford software.

The excitement that TEX and METAFONT generated among Donald Knuth's colleagues, and at the American Mathematical Society, led to a redesign and reimplementation of both in Pascal, released in 1982, and tweaked a bit in 1989. At the time, the only other widely-implemented programming languages were Fortran and Cobol, neither particularly suitable for writing typesetting software. Nevertheless, there was at least one early implementation of the SAIL version of METAFONT in Fortran [47], with the goal of producing fonts for Burmese.

By the late 1980s, the C programming language was becoming widely available: it became an ISO Standard in 1989. While C has a number of drawbacks, it has many fewer limitations than Pascal. A successful manual translation of TEX from Pascal to C by Pat Monardo at the University of California, Berkeley, about 1990, encouraged a collaborative effort on the Web2C translation system. Web2C recognizes just the subset of Pascal used in TEX, METAFONT, and their associated utility programs, and translates it to C. Today, most implementations are

Table 2: Notable historical events AC (after computers).

Year	Event
1940s	First digital computers
1968–1973	Niklaus Wirth invents Pascal language
1969–1970	Dennis Ritchie invents C language
1970s	`roff`, `script`, `runoff`, `document`
1975–1978	`eqn` (B. W. Kernighan and L. Cherry)
1976	`nroff` and `troff` (J. Ossanna),
1978	`bib` and `refer` (M. Lesk)
1977–1978	classic TEX and METAFONT in SAIL (D. Knuth)
1978–1980	SCRIBE (B. Reid)
1979	`tbl` (M. Lesk)
1981	`pic` (B. W. Kernighan)
1982	`ideal` (C. Van Wyk)
1982	'final' TEX and METAFONT in Pascal
1983–1985	LATEX (L. Lamport)
1984	BIBTEX (O. Patashnik)
1984	PostScript (Adobe Systems)
1986	`grap` (J. Bentley and B. W. Kernighan)
1989	'new' TEX and METAFONT (8-bit characters et al.)
1989–1991	HTML and HTTP at CERN (T. Berners-Lee)
1990	METAPOST (J. Hobby)
1991	World-Wide Web at CERN
1993	xmosaic browser (NCSA: M. Andreeson)
1993	PDF (Adobe Systems)
1994	LATEX 2$_\varepsilon$ (F. Mittelbach et al.)
1994	Ω (Y. Haralambous and J. Plaice) and Λ
1995–2000	WeBWork (University of Rochester)
1996	PDFTEX (Hán Thế Thánh)
1997	eTEX (P. Breitenlohner et al.)
1998	$\mathcal{NTS}$ (K. Skoupý)
2000	XMLTEX (D. Carlisle)
2001	JadeTEX (S. Rahtz)
2002	Donald Knuth celebrates $1{,}000{,}000_2^{\text{th}}$ birthday
2003	ant (ant is not TEX: A. Blumensath) (OCaml: 24K lines)
2003	Nottingham font conversion project (D. Brailsford)

based on the C translations, but the original Pascal source code remains definitive. System-dependent changes to the software are handled through change files that the tools `tangle` and `weave`, or their C equivalents, `ctangle` and `cweave`, turn into revised source code and documentation.

Karel Skoupý's $\mathcal{NTS}$ is a reimplementation of TEX in Java with the goal of improving modularization, and making possible experiments with new ideas in typesetting. Achim Blumensath's ANT (for ANT *is not TEX*) system has similar goals, but is done in the high-level OCaml language. Most of the coding of Ω, the extension of TEX for Unicode, is being done in C++, effectively a superset of C, and now about as widely available as C.

Although the Bell Laboratories' typesetting systems did not influence TEX very much, and were not available outside their development environment at the time that Knuth began his work on TEX and METAFONT in 1977, he was certainly aware of them [38, Chapter 2]. The Unix small-is-beautiful software-design methodology was similarly inapplicable on other operating systems, so TEX and METAFONT are each monolithic programs, of about 20,000 lines of prettyprinted Pascal code each. While relatively large programs at the time, they are dwarfed today by code projects that run to millions (e.g., GNU C compiler and library) and tens of millions (most modern operating systems) of lines.

What has set TEXware apart from most other software projects is the high degree of stability and reliability. Knuth attributes this to his development of *literate programming*[2] [54, 39, 37], which is so nicely illustrated in the TEX and METAFONT program source code [30, 32].

What we've accomplished

While TEX users are substantially outnumbered by users of desktop-publishing systems, TEX's open markup and document longevity, and its ability to handle mathematical and music markup, and scripts in many languages, and its possibility of identical output on all platforms from desktops to supercomputers, has ensured its continued use in some fields. In this section, we examine some of its achievements.

Books and journals More than a hundred books have been published about TEX and METAFONT,[3] and many thousands of books, and many journals, have been published with TEX acting behind the scenes as the typesetting engine.

[2] `http://www.math.utah.edu/pub/tex/bib/index-table-l.html#litprog`.

[3] `http://www.math.utah.edu/pub/tex/bib/index-table-t.html#texbook3`.

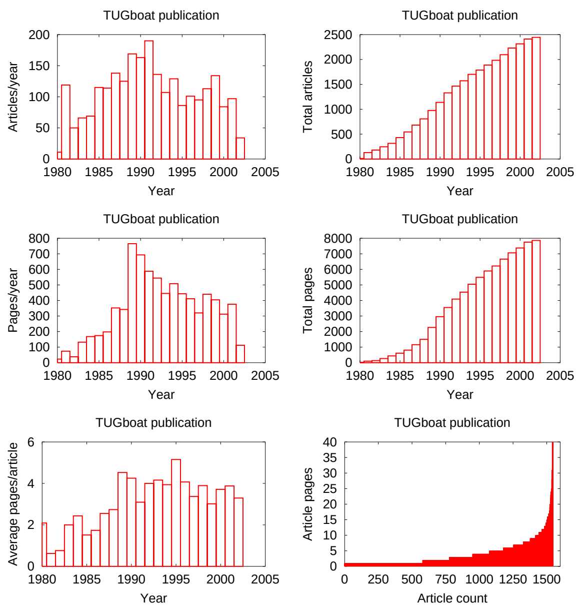


Figure 2: *TUGboat* publication statistics.

At least four journals, *TUGboat*, *Electronic Publishing—Origination, Dissemination, and Design*, *Markup Languages: Theory & Practice*, and *Serif*, have been devoted to typography, markup, and fonts; the bar charts in Figure 2 illustrate the activity in the first of these.

Today, many journals in computer science, mathematics, and physics use LATEX markup for author-submitted manuscripts. Sometimes, that material is converted by publishers into SGML or XML markup that TEX then typesets.

Several major publishers, including Addison-Wesley, Elsevier, Oxford, and Springer, have used TEX in book production.

Importantly for researchers, TEX markup has become a *de facto* standard in several technical fields, and because it requires nothing more than plain ASCII, it can even be used in e-mail messages.

Software archives There are huge archives of TEXware in the CTAN (Comprehensive TEX Archive Network) collections, with three master hosts,[4] and about 75 mirror sites around the world. *TUGboat* issues have at least twice been accompanied by CD-ROM copies of the CTAN archives.

The existence of many mirror sites makes it hard to document CTAN activity, but the logs from

[4] `ftp://ftp.dante.de`, `ftp://ftp.tex.ac.uk`, and `ftp://tug.ctan.org`.

just two of the master hosts record about 275,000 hits per week over the last seven years, from over a million Internet hosts.

In mid-2003, the CTAN archives contained nearly 80,000 files in about 6,000 directories, with close to 3,000,000 lines of class and style files. To put these numbers into perspective, Knuth's original `plain.tex` is only 1235 lines, and `manmac.tex`, which supplies additional macros needed for typesetting the TEXbook [29], is only 715 lines. Knuth's Computer Modern fonts are programmed in about 260 files, but the CTAN archives now hold nearly 6500 other METAFONT font programs. Wonderful, and skilled, volunteers did all of the rest of the work!

Document archives The rapid spread of the Internet led Paul Ginsparg in the early 1990s to create the Los Alamos archive of current research in high-energy physics. This archive is now hosted at Cornell University,[5] and access statistics show that at times, the archive has had more than a million hits a week. For many physics researchers, the archive, not print journals, *is* the current literature. Ginsparg was awarded a prestigious MacArthur Fellowship (worth US$500,000) in 2002 for this work.

The success of the physics archive has led to creation of similar projects in mathematics, nonlinear sciences, computer science, and quantitative biology, all reachable from links from the main archive Web site. Related efforts include the Networked Computer Science Technical Reference Library (NCSTRL)[6] and the Open Archives Initiative.[7] Collectively, these archives contains several hundred thousand research papers, most written in LATEX or TEX markup.

Bibliography archives In 1991, I began to record bibliographic data for my books and journals in BIBTEX markup. This evolved into the TEX Users Group Bibliography Project[8] covering the literature of much of computer science, electronic document production, and numerical mathematics. In 1995, we started the BibNet Project[9] with the more limited goal of recording complete publication bibliographies for leading researchers in numerical mathematics.

These collections now amount to more than 366,000 bibliographic entries in about 540 separate bibliographies, each of which is accompanied by additional files for spell checking, indexing, and typesetting the complete bibliographies. Because the data has been collected from many sources and extensively checked, it has considerably higher quality than most other collections. All BIBTEX files are prettyprinted, sorted, ordered, checksummed, and documented with a consistent comment header. Where possible, bibliographic entries contain hypertext links to the source of the data, and to online electronic documents.

In my department at Utah, the `bibsearch`[10] utility provides very fast access to these collections, plus another 203,000 in mathematical biology, 187,000 in computer science from the Digital Bibliography and Library Project (DBLP) at Universität Trier, Germany, and more than 1,261,000 in computer science from the world-wide computer science bibliography archive at Universität Karlsruhe. The two projects hosted at Utah are mirrored daily to the Karlsruhe archive. Archive statistics at Karlsruhe record about 300,000 hits per month, and at Utah, about 21,000 per month.

Both the American Mathematical Society MathSciNet database[11] and the European Mathematical Society E-Math database[12] now offer search results in BIBTEX markup.

The bibliography-archive work is supported by about 137,000 lines of code in the `awk` programming language, about 15,000 lines of code in Emacs Lisp, several thousand lines of Unix shell scripts, and several tens of thousands of lines of C code. Notable standalone tools in the collection include `bibcheck`, `bibclean`, `bibdup`, `bibextract`, `bibjoin`, `biblabel`, `biblex`, `biborder`, `bibparse`, `bibsearch`, `bibsort`, `bibsplit`, `bibunlex`, `citefind`, `citesub`, and `citetags`.

Many journal publishers now provide Web sites with publication data for recent issues. The JSTOR Project[13] provides complete coverage of all issues of about 360 journals in science and the humanities; it has allowed the preparation of complete bibliographies for the American Mathematical Monthly back to the first issue in 1894.

The increasing availability of publication data on the Web has made it feasible to develop tools

[5] `http://arxiv.org/`.

[6] `http://www.ncstrl.org/`.

[7] `http://www.openarchives.org/`.

[8] `http://www.math.utah.edu/pub/tex/bib/index-table.html`, `http://www.math.utah.edu/pub/tex/bib/idx/`, and `http://www.math.utah.edu/pub/tex/bib/toc/`.

[9] `http://www.math.utah.edu/pub/bibnet/`.

[10] `http://www.math.utah.edu/pub/mg/mg-1.3x/bibsearch/`.

[11] `http://e-math.ams.org/mathscinet/`.

[12] `http://www.emis.de/ZMATH/` and `http://zb.msri.org/ZMATH/`.

[13] `http://www.jstor.org/`.

for automatic conversion of such data, which tends to be quite similar across all journals from a single publisher. An essential tool for this work has been the HTML prettyprinter, `html-pretty`,[14] which allows fast, rigorous, and highly-reliable conversion of HTML documents to a standard layout of markup. Simpler tools, usually written in `awk`, can readily process that data to produce rough BibTeX markup that can be further cleaned up in a pipeline of some of the other tools listed above. A certain amount of manual cleanup is still required, notably bracing of proper nouns in document titles, but the bulk of the work has been done completely automatically.

Although the individual bibliographic and HTML tools are freely available, the Web page conversion software is not: it tends to require frequent maintenance as whimsy overtakes good sense at publisher Web sites, and in any event, is really needed only at the site doing the HTML-to-BibTeX conversion work.

I have collaborated with several publishers and journal editors, so that these BibTeX archives receive prominent links from their journal Web sites, and so that I get prompt notification of the appearance of new journal issues. In extremely favorable cases, the BibTeX data can be available about ten minutes after a journal issue announcement.

This bibliographic activity is a substantial effort on my part, but the reward is that the entire Internet community gets quick access to the data. Also, BibTeX, LaTeX, and TeX get free advertising in communities that might be entirely unaware of them, and I can at last find material in the thousands of journal issues on my own shelves.

BibTeX markup is extremely flexible, even if the associated style-file language is somewhat arcane. Publisher interest in XML markup led to the BibTeXML project at the Swiss Federal Institute of Technology (ETH) in Zürich, Switzerland, which has developed software for conversion between XML and BibTeX markup of bibliographic data; at the time of writing, the project Web site is not accessible. Because XML is not extensible at the document level, such a conversion is not as simple as it might first appear.

What did TeX do right?

Twenty-five years is a very long time in the rapidly-developing computing industry, and it is appropriate to look back over the use of TeX in that time, and comment on its successes, and its failures.

[14] `http://www.math.utah.edu/pub/sgml/`.

While a list of such points is necessarily a matter of personal opinion, I believe that it is worthwhile to enumerate the ones that I have found significant. Although I concentrate mainly on TeX, some of the remarks should be interpreted to include the associated utilities that I call TeXware.

Open software The most important done-right feature of TeX is that it is an *open-source literate program.* Without quibbling over the exact meaning of open source, the essential point is that *anyone* can use TeX for any purpose, commercial or noncommercial, as long as they don't change the non-system-dependent parts of it without changing its name. One of the TUG'2003 conference speakers, Ajit Ranade, noted that TeX is responsible for a multimillion dollar typesetting-support industry in India that employs thousands of people.

This openness should be contrasted with the abysmal state of the desktop-publishing industry where markup is generally keep secret and proprietary, holding user documents hostage to marketing whims of software vendors, and making it impossible to achieve consistent output when documents are moved between platforms, or to archive documents for long-term access. This deplorable situation is getting worse, not better: one desktop-publishing vendor is moving towards encrypted file formats that can be decoded only by that vendor's products; the marketing justification is called *document security*, but it also locks out competition and gives the vendor, not the user, control over document access.

Typesetting kernel TeX provides a small kernel of primitives that are specialized for the complex, and often idiosyncratic, job of typesetting. The best-known analogue of this software model is the PostScript page-description language, which provides primitives that are highly suited to placing marks on a page, assuming that some other software package has first decided where they go.

Extensible typesetting language Although most of TeX's typesetting kernel is rather low level, dealing with object positioning and selection, through a powerful, albeit arcane, macro language, TeX can be extended to provide higher-level markup. Such extensions significantly lessen the demand for changes in the underlying software, shielding it from the creeping featurism that plagues most commercial software, and prevents reliability and stability.

The most successful of these extensions is the LaTeX document preparation system [41, 42] and the many packages built on top of it [16, 17, 18]. LaTeX, like BibTEX, is strongly influenced by Brian Reid's pioneering Scribe system.

The key feature of LATEX markup is that typesetting objects such as title and author information, tables of contents, abstracts, chapters, sections, subsections, equations, figures, tables, glossaries, indexes, and so on, are general concepts shared by most documents, and are thus marked up with commands that tell *what to do*, rather than *how to do.* This is usually termed *logical markup*, as opposed to *physical*, or *visual, markup.* Most desktop-publishing software is modeled on physical markup: the what-you-see-is-all-you've-got approach is burdensome on the user, and makes it nearly impossible to achieve formatting consistency.

In principle, the meaning of logical markup can be changed simply by declaring a new document class at the start of the top-level file. In practice, other changes may be required as well, but they are usually limited to issues of front-matter markup, which can be quite complex for some documents, and to issues of how literature citations are used in the text (e.g., the numbered style used in this article, versus the author-date style common in the humanities). Nevertheless, the changes needed to switch document style are generally a small part of the total effort of document production.

The need for presentation of the same information in different formats is often not appreciated by authors, but publishers at past TEX Users Group conferences have reported that they have sometimes been able to reuse information in a dozen ways, each of them generating income. Document reuse is a major driving force in the use of SGML and XML markup for long-term document storage in the publishing industry. For example, while most article submissions to the American Chemical Society (the world's largest publisher of chemistry literature) are in word-processor formats, all submissions are converted to SGML with a combination of in-house format-conversion software and manual touch-ups.

Device-independent output The Bell Laboratories' `troff` system produced output for one particular typesetting device, now long defunct, and its design was heavily influenced by the capabilities and limitations of that particular device.

TEX's output pays homage to no particular device: it is a compact device-independent format, called a *DVI* file. The job of converting that file for any particular output device is left to separate software, called a DVI driver. This task is far from trivial: current drivers for single devices are 22,000 (`dvips`) to 29,000 (`xdvi`) lines of code, and my own DVI driver family, which supports dozens of devices, is about 97,000 lines. These are all larger than TEX itself. Popular output formats have evolved during TEX's life, but TEX remains blissfully independent of that evolution.

Current analogues of TEX's design choice are the virtual machine definition underlying the Java and C# programming languages, and the virtual machine layer provided on IBM mainframes since the early 1970s. More recently, the VMware[15] system on the Intel IA-32 platform permits multiple operating systems to be simultaneously active on the same hardware, and IBM PowerPC systems now support sharing of CPUs by separate operating systems.

Although they were not part of the original design of TEX, Geoffrey Tobin's excellent `dv2dt` and `dt2dv` tools included in many modern TEX distributions provide a way to convert the compact DVI format to a human-readable, and thus, editable, form, and back again.

Font independence The lack of good-quality vendor-independent fonts forced Donald Knuth to develop an outstanding font-design system, METAFONT. However, he was careful to isolate most of the details, so that TEX only needs to know about the *dimensions* of the fonts, through the compact *TEX font metric* (TFM) files. TEX remains completely unaware of character shapes and how they are represented in font files.

This independence has proved a great virtue, allowing TEX to use almost any font, subject to an unfortunate limitation on the number of characters in a font, provided that metrics are available. Some commercial font vendors in the early days of TEX would not release that information, and some still do not permit its free distribution.

With virtual-font technology [26], composite fonts can be created whose glyphs come from other fonts, even other virtual fonts. This makes it possible, for example, to remap glyphs into the order expected inside TEX, avoiding the need to redefine numerous macros that assign names to font characters. PostScript Type 1 fonts are generally accessed via virtual fonts.

Open font specification Unlike most commercial fonts of the 1970s, METAFONT's output file formats are openly, and well, documented. Besides the TFM file of font metrics, METAFONT produces a file of character bitmaps, called a *generic font* (GF) file.

When superior font compression techniques were developed by Tom Rokicki in 1985 [52], he was able to write a clean font-format translation tool,

[15] `http://www.vmware.com/`.

gftopk. Today, most TEX sites store only the more compact PK format.

Boxes, glue, and penalties A great insight in the design of TEX is the concept of boxes of text, and flexible space between them, called *glue* (though *springs* might have been a better characterization). TEX builds up lines, paragraphs, and page galleys as lists of boxes separated by glue, where the glue has user-defined stretchability and shrinkability.

Centered, ragged-right, and ragged-left typesetting are all straightforward to implement with glue.

For further control, users can insert penalties at suitable points to encourage or discourage line breaks and page breaks. TEX can then apply a mathematical optimization technique to determine the best way to typeset pages; few other systems, before or since, do as good a job.

Compact markup for common cases The first thing that one sees when comparing SGML or XML markup with TEX markup is that the first two are painfully verbose. TEX makes common cases simple and compact. For example, instead of wrapping paragraphs with SGML commands `<paragraph>` ... `</paragraph>`, TEX simply assumes that a line that is blank or empty separates paragraphs. When this is not convenient or possible, it offers the `\par` control word to mark the boundary.

Other examples of TEX's brevity are the use of braces for grouping, dollar signs around mathematics markup (Knuthian humor: mathematics was traditionally expensive to typeset), and caret and underscore for mathematical superscripts and subscripts.

Consider the lowly, but important, period (dot or full stop): it is the smallest character in a font, but it can mean end of initial, end of abbreviation, end of sentence, decimal point, Internet hostname separator, filename separator, or be part of an ellipsis. TEX has a simple rule for its interpretation when it is followed by an input space: if it follows a capital letter, it ends an initial, and otherwise, it ends a sentence. This heuristic is almost always correct, but careful LATEX typists will write `Back in the USSR\@.` to instruct TEX that an intersentence space, rather than an interword space, is called for, or use a tie (~) or literal space (`\␣`) when only an interword space is needed. The special spacing required for ellipses are handled by standard control words: `\cdots`, `\ldots`, and `\ddots`.

Nonsignificant spaces In `troff`, spaces are significant: two spaces in the input produces two spaces in the output, plus or minus a bit to justify lines. Consequently, indentation cannot be used in `troff` input to improve readability and highlight nested document structure.

TEX avoids this significant design flaw by treating a sequence of spaces as equivalent to a single space. Similarly, any sequence of empty or blank lines is equivalent to a single paragraph break. When this behavior is unwanted, TEX offers verbatim environments.

Identical results on all systems A huge virtue of TEX is the possibility of getting identical output on all platforms. Indeed, as long as all of the input macro packages and font metrics are identical, output *is* identical. No commercial desktop-publishing system even comes close.

TEX achieves this platform-independence by carrying out all arithmetic identically: it uses exact fixed-point arithmetic in all computations that affect line and page breaking. For dimensions, an underlying 32-bit integer word is split into fragments $1 + 1 + 14 + 16$: a sign bit, an overflow bit, a 14-bit integer part ($2^{14} = 16384$), and a 16-bit fractional part. The largest dimension is about the width of a room, and the smallest spacing is less than the wavelength of visible light: computational rounding errors are invisible in the output. TEX also supports pure 32-bit integer arithmetic in computations with `\count` registers.

Dimension independence Systems of measurement differ between different cultures and countries: TEX allows dimensions to be specified in any of nine different units that cater to the majority of needs.

Dynamic loading of files TEX permits temporary redirection of its input stream to another file, via the `\input` control word. That file can in turn contain other `\input` commands, with the result that packages of commonly-used commands are easily supported, and users can break up large documents into manageable pieces. This saves processing time, since only those pieces being worked on need to be typeset, and also reduces the effect of global editing disasters.

Redefinition of character meaning TEX assigns each input character one of sixteen category codes that controls further interpretation of each character, and those assignments can be changed at any time via the `\catcode` command. Although few users directly exploit the feature, its existence makes it possible for TEX to typeset documents written in quite different markup, such as control words that begin with `@` in Texinfo, and angle-bracket delimited words in SGML and XML (e.g., `jadetex` and `xmltex`).

No system call Because TEX is expected to run on many different platforms, it does not offer any internal way of communicating with the underlying operating system, such as via a `\system{...}` command.

While such an ability can be handy, allowing, for example, a book index to be prepared immediately before it is typeset, experience on the Internet has shown that such power is too frequently exploited for nefarious purposes. The words *worm* and *virus* are now familiar to the general population, even those who have never used a computer. When the mere viewing of a document can cause arbitrary commands to be executed, security and stability are impossible.

Last definition holds Although it seems trivial, in TEX, as in most programming languages, the last definition or assignment of an object is the one that is used.

SGML, by contrast, uses a first-definition-holds rule. Instead of being able to load up a base package of command definitions, and then make minor tweaks to it by subsequent redefinitions and assignments, each package modification must be defined as a complete new package, with altered definitions preceding an inclusion of the original package.

`\special` command In order to allow support of unforseen features, TEX provides the `\special` command whose argument is recorded verbatim in the DVI file. Graphics, color, and hypertext links are common examples that use `\special`. Of course, DVI drivers then need to be taught how to handle such material.

Stability and reliability TEX is quite possibly the most stable and reliable software product of any substantial complexity that has ever been written by a human programmer. Although its development has not been free of errors, most TEX users have never seen an error, much less a crash, in TEX.

What errors have been found have been well studied and documented in *The errors of TEX*: see [35] and an update in [37, pp. 243–339].

Illustrations by Duane Bibby[16] In a remarkable collaboration that has lasted nearly two decades, the gifted illustrator Duane Bibby has worked with members of the TEX community to prepare wonderful drawings for not only Knuth's *Computers and Typesetting* series, but also for several LATEX books, and numerous T-shirts and mugs at TEX Users Group conferences.

It was such a pleasure for many of us to meet Duane for the first time at this meeting, to hear his talk on his long collaboration with Donald Knuth, and to see how the TEX lion and METAFONT lioness evolved.

Duane Bibby's drawings, and Donald Knuth's wit and superb writing skill, add light and humor to what might otherwise be a dry and daunting manual on the complex subject of typography.

[16] This section of my address was written long before I found out that Duane would be attending the conference.

What did TEX do wrong?

It is now time to drop our laudatory stance, and become a grinch: nothing is perfect, not even TEX, and with 25 years of hindsight, it is time to assess what it did wrong.

There has been a lot of computer technology developed since 1977 that few could have predicted then, including personal computers, high-quality printers, PostScript, PDF, window systems, and the World-Wide Web. Hardware costs have dropped, and speed and capacity have grown, at a pace that is unparalleled in the history of human development. Most of the criticisms of this section would have been both unfair and unforseen when TEX was first designed.

No rigorous grammar The biggest deficiency in TEX that has always bothered me is that it is not based on a rigorous programming-language grammar. This is particularly puzzling when its author is the founder of modern LR parsing [28], and that work is cited among the great papers in computer science [43]. When I asked Don about this once, he jokingly responded that he didn't believe in grammars!

Most programming languages designed since Algol 60 have been based on grammars, but TEX is not. Lack of an independent grammar means that the only truly reliable parser of TEX input is TEX itself, yet long experience with other programming languages has shown that rigorous grammars can lead to highly-reliable machine-generated parsers (Unix `yacc`, Berkeley `byacc`, GNU `bison`, and Holub's `occs` [23] are good examples of parser generators). Parsers are needed not only in compilers, like TEX, but also in prettyprinters, syntax checkers, tag extractors (`ctags` and `etags`), and other tools that operate on the input language for purposes other than its execution.

Precisely because human programmers are fallible, it is important to have multiple independent implementations of any significant program, and to have implementations available on several machine architectures. Only when those programs produce identical output can one have any confidence in their results. Languages like `awk`, Cobol, Fortran, C, C++, Common Lisp, and Java enjoy this diversity, while others, like `axiom`, C#, `delphi`, `maple`, `mathematica`, `perl`, `python`, `ruby`, `reduce`, `sas`, `spss`, `tcl`, and Visual Basic, do not. Since modern compilers are usually very much bigger than the programs that they process, when a bug or other unexpected behavior surfaces, it is legitimate to ask whether the bug is in the compiler, or in the program. If the misbehavior disappears when the compiler or platform changes, suspicion rests on the compiler.

The lack of diversity for TeX has been less of a problem, simply because of the enormous talent behind it. Nevertheless, it is good to see the appearance of $\mathcal{NTS}$ as an independent implementation of TeX.

Macro, not programming, language TeX's extension language is a macro language, not a proper programming language. Every TeX macro programmer who has implemented a nontrivial operation has suffered from the difficulty of TeX's macro language. Things that are simple to do in other languages designed for text processing are frequently very painful in TeX. Macro languages have too many side effects, and lack the power of true programming languages.

Lisp programmers would argue that the proper approach is to eliminate that separation between programs and data: making data look like programs means that the data can *be* a program, and that has been found to produce great power and generality. Luigi Semenzato and Edward Wang at the University of California, Berkeley, investigated a Lisp-like interface to TeX [53].

Too much hard coded The limited 18-bit address space of the 36-bit DEC PDP-10 architecture[17] of TeX's original development platform, and severe limitations of the Pascal programming language, constrained many aspects of the program.

There are many examples of objects inside TeX whose size is fixed when TeX is compiled for a particular platform: the dreaded `TeX capacity exceeded` message is familiar to every serious TeX user.

Fortunately, in recent years, the Web2C implementation has replaced many of these fixed limits by configurable limits settable in startup files, although other implementations may not be so flexible. Nevertheless, modern programming practice in the GNU system and others is that software should have no hard limits, other than those found to be available at run time, or enforced by the underlying architecture.

It isn't just table sizes that are hard coded in TeX: many algorithms are too, notably, hyphenation, line breaking, page breaking, and float placement. While TeX provides powerful ways to influence these algorithms, the algorithms cannot be ripped out and replaced dynamically at run-time, and they cannot be changed in TeX itself without producing a program that is no longer allowed to call itself TeX.

Too many fixed-size objects The need to squeeze a large program and data into the small PDP-10 address space led to further economizations in TeX that are difficult to remove, because the sizes of various objects themselves are encoded in bitfields of other data structures. If only 8 bits are available for the size, then only $2^8 = 256$ objects can be created. This limitation is seen in the number of various types of TeX boxes, category codes, token lists, registers, and skips. Worse, it permeates TeX's internal source code, making it very hard to eliminate.

Enlarging these limits was one of the major design goals of ε-TeX, which is extended from TeX with a change file that is about a third the size of TeX itself. For comparison, pdfTeX augments TeX's DVI output with a completely new, and very complex, output form: PDF; its change file is about 40% the size of TeX.

Too many global variables In the 60-year history of computing, program after program has foundered in complexity arising from too much global state. When code depends on global variables that can be changed arbitrarily at any place in the code, it becomes impossible to reason about the correctness of the program. Redesigns of the relatively small and simple Unix operating system in the form of Mach, Solaris, and GNU/Linux all attempted to sharply limit the problem of global state, by repackaging the software into independent layers with simple, and well-defined, interfaces.

TeX has too many global variables and macros that can be changed anywhere, at any time, by any user.

Too little tracing For debugging, and for understanding unfamiliar packages, it is desirable to be able to request tracing of the use and (re)definition of commands, files, and registers. Although TeX provides a few tracing commands, their output usually overwhelms the user, because there is no way to restrict the tracing to a specified set of names. Also,

[17] In modern terms, 1.3M characters or 0.25M words.

TEX provides no way to record *where* definitions happen, yet precisely that information is needed to prepare customizations.

Name collision Older programming languages, such as Algol, Fortran, Cobol, Pascal, and C all have very limited control over name visibility. Usually, this amounts to just *local* variables (defined in a function or subroutine), and *global* variables, known throughout the program. C adds a bit more control in offering *file-global* variables that are known throughout a single source file, but inaccessible elsewhere.

Descendants of these languages, such as Ada, Fortran 90, Cobol 2002, Modula, C++, Java, and C# all introduce additional constraints through modules or namespaces to allow compartmentalization and control of names.

Sadly, TEX has effectively only one level of naming: global. Although names can be defined inside braced groups, the size of groups can be severely constrained by internal buffer sizes, and in any event, groups are anonymous: you cannot refer to names in other groups except by nasty or tricky subterfuges that make program maintenance impossible.

Lisp too had this defect, but it became common practice to use long descriptive names that incorporate a package prefix, such as `LaTeX-pageref-with-completion` from my LATEX editing-support package for Emacs. However, with default category codes, TEX limits command names to just the Latin letters, so the only way to avoid inadvertent collisions with names in other packages is to use long names, and the only way to make them readable is to use mixed capitalization, such as `\DeclareRobustCommand` in the LATEX kernel.

The lack of hooks into the entry and exit of commands means that packages are often forced to redefine macros used by other packages. Loading of multiple packages sometimes results in puzzling, and hard-to-debug, interactions of these redefinitions.

Inadequate I/O Like Fortran, TEX's I/O model is based on lines, rather than characters, with the additional restriction that braces (or characters of that category) must be properly nested. By contrast, the C programming language provides `getc()` and `putc()` primitives to read and write single characters, and I/O of higher-level objects can be built from these alone.

Java, C++, and C# go a step beyond C in generalizing I/O to a stream of data in which processing filters can be arbitrarily, and transparently, inserted. The streams need not even be directed to and from physical files, but can instead refer to strings in memory, or network devices, or display devices, or virtual files.

Unlike most other programming languages, TEX does not offer anything analogous to Fortran's formatted I/O or C's equivalent of `fscanf()` and `fprintf()`, which provide detailed control over the interpretation of input, and the appearance of output. A satisfactory I/O library is astonishingly complex: it deserves to be rigorously defined, standardized, and incorporated in every implementation of the programming language.

The deceptively-easy task of expressing binary fractional numbers as human-readable decimal numbers led TEX's author to write a famous paper called *A Simple Program Whose Proof Isn't* [36]. Related articles that finally properly solved the number-base conversion problem have appeared only since 1990 [1, 7, 20, 55, 56].

Character set limits TEX processing is based entirely on a model of characters that can be represented as a single 8-bit byte, in the ASCII encoding. This was not much different from other software of the 1970s, and at least, TEX could handle lowercase letters, and with the help of control symbols and control words, even decorate letters with accents.

However, it was clear that $2^8 = 256$ characters are not enough, not even for European use. ISO has already standardized ten different ISO8859-*n code pages* based on the Latin alphabet, with a few more to support Arabic, Cyrillic, Hebrew, and Thai.

Two separate efforts to standardize much larger character sets began in the early 1990s: Unicode and ISO 10646. After some divergence, the two groups are fortunately now coordinated, with Unicode guaranteed to be a strict subset of ISO 10646. Several operating systems have adopted Unicode as their standard character encoding.

Unicode developers hold that 21 bits are sufficient to encode all real writing systems in human history. This is a repertoire of about two million characters, of which 96,283 are encoded in the latest version [58]. Of these, 70,203 are ideographic characters from Chinese, Japanese, Korean, Yi, and historical Vietnamese [44, 45].

The large number of Unicode characters means that neither 8-bit nor 16-bit values suffice. 24-bit words are impractical in current computer architectures, and native 32-bit words waste a third of the bits. In practice, then, Unicode is represented in one of several encodings that require one or more 8-bit or 16-bit chunks to represent a single character. One of these, UTF-8, contains ASCII and most ISO8859-1 characters in their normal positions, but uses up to four bytes for some other characters. UTF-8 was developed at Bell Laboratories for the Plan 9 operating system, and because most of the world's existing computer files are in either ASCII or ISO8859-1, they are already Unicode-conformant when interpreted in the UTF-8 encoding.

Existing programming languages have usually been defined with the assumption that a single character can be held in an 8-bit byte, and the effects of that assumption are deeply rooted. Most languages are only beginning to address the problem of how to deal with Unicode data, and none of the initial attempts, in C, C++, C#, and Java, are yet very satisfactory.

TeX too requires massive changes to handle Unicode, and although the work on Ω was originally based on a change file for TeX, its developers report that a completely new program, in C++, will probably be needed.

No input filters The code-page morass mentioned in the previous subsection has an immediate impact on TeX documents written in a variety of European languages. In particular, the encoding must be known, and then translated to TeX's internal expectations, before TeX can process the document. Filesystems rarely record character-set information, so documents have to do so themselves.

TeX processing would be easier if it had input filters that could transparently supply the needed translations. It would have been relatively easy to implement them by making the internal `xchr[]` array [30, §21, p. 10] accessible to the TeX user.[18] Instead, TeX erroneously assumes that character sets are a unique property of each platform, and therefore, can be hard-coded into the implementation on each system. That assumption was correct for EBCDIC on IBM mainframes versus ASCII everywhere else, but it was false as soon as code pages were introduced.

The need for input translation is so strong that Ω developers from an early stage in its design introduced Ω *translation processes* (OTPs).

No color state In 1977, there were no low-cost color output devices, and some commercial typesetting systems were incapable of handling color. Consequently, TeX is completely ignorant of color. However, color is a text attribute much like the current font, although it can also be a page-background or region-shading attribute. In particular, once set, the current colors should remain in effect across line breaks and page breaks, just like the current font does.

[18] Some implementations have extended TeX to provide such access.

In order to make it possible to process selected pages of DVI files, TeX records in the DVI postamble a list of all required fonts, and at the start of each page description in the DVI file, it records the current font, so that the page can be rendered independently of all preceding pages.

In the same way, the current colors are needed at page start. Since TeX doesn't provide for color, its support has to come through the `\special` command. However, TeX lacks a hook (code fragment) to be executed when a page is shipped out to the DVI file (actually, it does, but output routines are very fragile, and depend on the macro package in use), so there is no clean way to record the current colors on each page. This means that DVI drivers that support color are now forced to read the entire DVI file, parse all of its `\special` commands, and build up a list of starting colors for each page, even if the user just wants to display a single page. Fortunately, modern machines are fast, so most users probably never notice the delay.

No graphics In the 1970s, commercial typesetters had no graphics capabilities. Although researchers in computer graphics had produced a rudimentary draft graphics standard [2] a few months before work began on TeX, it took two more years for a more detailed draft standard [3], and much of that specification could not be implemented portably. Sadly, graphics standardization splintered and foundered in the 1980s, and today, the ISO graphics standards that exist are largely ignored by programmers.

Nevertheless, a small number of graphics operations, notably dot and line primitives, an elliptical arc primitive, and a region fill, would have been sufficient to represent most technical drawings. They could have been compactly encoded in the DVI file, and reasonably easily translated by DVI drivers.

LaTeX's `picture` mode can emulate curves by drawing tiny squares along the curve path, but their internal representation is extremely inefficient, and even a few such curves soon exhaust TeX's memory.

Today, most graphics in TeX documents is represented by PostScript (released in 1984) figure inclusions.

One page at a time Memory constraints forced TeX to handle page creation by building up a sequence of objects on the main vertical list, asynchronously invoking an output routine as the list contents get big enough to fill a single output page.

However, in high-quality manual typesetting, page-layout design proceeds in pairs of facing pages. With a lot of work, it would be possible to program a TEX output routine for handling pages in pairs, but the task is decidedly nontrivial, and in LATEX, the output routine is viewed as a fragile part of the kernel that cannot be altered without great risk of breaking other parts of LATEX.

Multicolumn deficiency Multicolumn output is challenging, for three main reasons:

1. Narrow columns make it difficult to maintain right-margin justification without excessive hyphenation, or objectionable whitespace. Every reader of a printed newspaper is familiar with these problems.
2. Esthetic considerations require columns to be balanced, perhaps all with the same length, or with all but the last of uniform size.
3. Complex documents may require changing column formatting within a single page. In these proceedings, for example, article front matter is in one-column mode, but article text is in two-column mode. Some American Physical Society journals have particularly complex column-formatting practices, and newspaper layout is even more difficult.

Although it has been possible to prepare style files to support multicolumn typesetting, such as Frank Mittelbach's `multicol` package [46] and David Carlisle's and Arthur Ogawa's REVTEX package [49] designed for the American Physical Society, and also style files to support the flowing of text around figures [16, §6.4], only a handful of TEX wizards are capable of creating such packages. I suspect that all of these packages can be broken in certain cases: while they may be quite good, they are not robust.

Had TEX been designed from the beginning to typeset into a list of regions of arbitrary user-specifiable shapes, instead of into a single rectangular page, even the most complex magazine and newspaper layouts could readily be accommodated.

As the doctoral work of Michael Plass [40, 50] and Stefan Wohlfeil [60] has shown, line-breaking and page-breaking algorithms are extremely difficult. In my view, they should be implemented in a dynamically-loaded module in the programming language that TEX doesn't have.

Not general enough for all writing directions TEX expects text to be laid out horizontally from left to right. While this works for many languages, it doesn't handle right-to-left Semitic languages, or vertical writing directions (left to right across the page, or the reverse) needed for Chinese, Japanese, Korean, and Mongolian.

Donald Knuth and Pierre MacKay were able to extend the TEX program to handle mixtures of left-to-right and right-to-left text [27], producing the TEX--XET derivative, and ε-TEX now includes that extension. In Semitic languages, numbers are written from left to right, so those languages always need bidirectional typesetting. Similar extensions have allowed TEX derivatives to handle some of the vertical typesetting needs in East Asia, although I suspect that Mongolian remains unsupported.

In the `troff` world, Becker and Berry [5] were able to extend that program for tri-directional typesetting, but could not handle a fourth writing direction.

In our global community, a typesetting system must be able to handle all traditional writing directions, including mixtures of all of them in a single document (e.g., a Chinese literary article citing English, Arabic, and Mongolian texts).

No DVI output pipe TEX expects to write an entire DVI file under its exclusive control, relinquishing it to others only when TEX is done. Unix pipelines demonstrate that it is very useful to sample program output before it is complete, but Unix was only just emerging from Bell Laboratories when TEX design began.

In the early 1980s, in unpublished work demonstrated at early TEX Users Group conferences, David Fuchs at Stanford University ported TEX to the cramped and crippled environment of IBM PC DOS, and then in a tremendous feat, extended TEX to allow immediate communication with a screen previewer. His work became a commercial product, MicroTEX, but its vendor shortly thereafter withdrew it from the market.

Later, Blue Sky Research produced Lightning TEXtures on the Apple Macintosh, which retypesets the document as changes are made in its input files.

More recently, Jonathan Fine in Cambridge, UK, developed `texd` [12], a permanently-running resident daemon TEX that can be called upon at any time to typeset a fragment of TEX code, and return DVI code.

Certainly, TEX's descendants should learn from these extensions.

No sandbox When TEX was born in 1977, the Arpanet was eight years old, but still an infant, with fewer than a hundred hosts and a few thousand users with a strong community spirit. Today, the Internet has hundreds of millions of hosts, and is a decidedly hostile environment.

TeX does not offer a way of restricting its actions to benevolent ones. Although we noted earlier that TeX cannot run external programs, it can nevertheless (over)write a file anywhere in the file system that the user has write access to.

PostScript also has this problem, but the designers of the `ghostscript` implementation of that language added the `SAFER` option to limit filesystem access.

The TeXlive implementation of TeX contains changes to restrict output to the current directory, and prevent writing special configuration files that might open security holes, but most other implementations lack these features.

Uncaught arithmetic overflow To enhance reliability, TeX catches arithmetic overflow from multiplication, but curiously, not from addition [30, §104], which you can readily demonstrate with this small example:

```
\count0 = 2147483647
\advance \count0 by \count0
\message{ADD: count0 = \the \count0}

\count0 = 2147483647
\multiply \count0 by 2
\message{MULTIPLY: count0 = \the \count0}

\bye
```

TeX reports in the output log:

```
ADD: count0 = -2
! Arithmetic overflow.
l.8 \multiply \count0 by 2
MULTIPLY: count0 = 2147483647
```

When I asked Don about this, he responded that there were too many places in TeX where integer addition was done. In my view, that is simply a design flaw: all of those additions should be done in one function that is called wherever needed. It is much better to get the right answer a little slower, than to get the wrong answer fast.

TeX is not alone in partly, or wholly, ignoring integer overflow: many CPU architectures, operating systems, and programming languages do too. Several major civilian, government, and military disasters have subsequently been attributed to arithmetic overflow [48].

32-bit precision too limiting Although TeX's design choice of using fixed-point arithmetic was critical in achieving its goal of identical results everywhere, there are applications where 32 bits are insufficient. One of them is the implementation of trigonometric functions needed for computing text rotations, and another is fixed-point division.

No floating-point arithmetic When TeX was designed, floating-point arithmetic systems varied widely, with 24-bit, 32-bit, 36-bit, 48-bit, 60-bit, and 64-bit sizes on various platforms. Worse, their implementations were sometimes seriously flawed, with anomalies like $(z + z) \neq 2 \times z$, $z \neq 1 \times z$, $y \times z \neq z \times y$, and **if** $(z \neq 0.0)$ $x = y/z$ terminating with a zero-divide error. It would thus have been untenable for TeX to use native hardware floating-point arithmetic for calculations that affect output appearance.

Starting in the late 1970s, a much-improved floating-point arithmetic system was designed by an IEEE working group. Although the system was not standardized until 1985 [24], it was first implemented in the Intel 8087 already in 1980. This IEEE 754 floating-point arithmetic system has been part of almost every new computer architecture designed since then.

Despite standardization, variations in internal details of specific hardware implementations of IEEE 754 prevent getting identical results everywhere. However, but for TeX's seniority, TeX could have had its own *software* implementation of IEEE 754 arithmetic that behaved identically everywhere, and it could have had its own repertoire of elementary functions (trigonometric, logarithmic, exponential, and so on) that would have greatly simplified some typesetting applications.

No conventional arithmetic expressions TeX's commands for integer arithmetic, illustrated in the overflow example given earlier, are inconvenient: it would have been much better to have a conventional arithmetic-expression facility. It is possible, though difficult, to do so in TeX's macro language [19]. The LaTeX `calc` package only provides arithmetic expressions in a limited context.

Expression parsing of one of the first examples given in books on compiler design, and is not particularly difficult to implement; it should have been in TeX from the beginning.

No word and line boundary markers With any document formatting or markup system or typesetting system, one sometimes needs to extract text from the formatted output, perhaps because the input is not available, or cannot be deduced from the input without implementing a complete parser.

While input usually reflects output, this need not be the case, as shown by David Carlisle's seasonal puzzle [8]. His plain TeX input file from the CTAN archives[19] is reproduced in Figure 3, and has no apparent relation to the typeset output, a well-known poem.

```
\let~\catcode~`76~`A13~`F1~`j00~`P2jdefA71F~`7113jdefPALLF
PA''FwPA;;FPAZZFLaLPA//71F71iPAHHFLPAzzFenPASSFthP;A$$FevP
A@@FfPARR717273F737271P;ADDFRgniPAWW71FPATTFvePA**FstRsamP
AGGFRruoPAqq71.72.F717271PAYY7172F727171PA??Fi*LmPA&&71jfi
Fjfi71PAVVFjbigskipRPWGAUU71727374 75,76Fjpar71727375Djifx
:76jelse&U76jfiPLAKK7172F71l7271PAXX71FVLnOSeL71SLRyadR@oL
RrhC?yLRurtKFeLPFovPgaTLtReRomL;PABB71 72,73:Fjif.73.jelse
B73:jfiXF71PU71 72,73:PWs;AMM71F71diPAJJFRdriPAQQFRsreLPAI
I71Fo71dPA!!FRgiePBt'el@ lTLqdrYmu.Q.,Ke;vz vzLqpip.Q.,tz;
;Lql.IrsZ.eap,qn.i. i.eLlMaesLdRcna,;!;h htLqm.MRasZ.ilk,%
s$;z zLqs'.ansZ.Ymi,/sx ;LYegseZRyal,@i;@ TLRlogdLrDsW,@;G
LcYlaDLbJsW,SWXJW ree @rzchLhzsW,;WERcesInW qt.'oL.Rtrul;e
doTsW,Wk;Rri@stW aHAHHFndZPpqar.tridgeLinZpe.LtYer.W,:jbye
```

Figure 3: David Carlisle's seasonal puzzle file for plain TEX.

Unix tools such as `antiword`, `dehtml`, `deroff`, `detex`, `dexml`, `dvi2text`, `dvi2tty`, `pdftotext`, `ps-2ascii`, and `pstotext` attempt to do this text extraction for common document formats, but none is entirely successful, because they all have to apply fragile heuristics to deduce word and line boundaries from spacing.

Becker and Berry [5, p. 134] pointed out that they could not apply their method for tridirectional text to TEX because of its lack of line-boundary markers.

When text is typeset, TEX usually knows where each word begins and ends, and where it has used hyphenation for improved line breaking; sadly, this important information is lost in the DVI output.

Given the complexity of deciphering TEX input, checks for spelling, doubled-word, and delimiter-balance errors really should be done on text extracted from the DVI file.

No paper size TEX's view of a page is a galley of material of unknown width and indefinite height, measured in a left-handed coordinate system beginning at an offset from the top-left corner of the page. The main body of each page is expected to fill a rectangle of size `\hsize` $\times$ `\vsize` (in LATEX, `\textwidth` $\times$ `\textheight`). The corner offset of its $(0, 0)$ point is, regrettably, a parochial one inch from each edge.

TEX never admits that there might be a physical page of one of several common standard sizes and names (e.g., A, A4, quarto, foolscap, JIS-B5, ...), and consequently, users who want to adjust their text dimensions to suit a particular paper size may have to tweak several different dimension parameters.

No absolute page positioning Because of the developing page galley, within the input document, there is no way to refer to an absolute position on the output page, such as for typesetting a company logo, or displaying a red-lettered document-security classification. The only way that this can be achieved is to hook into the fragile output routine, most safely by attaching the desired material to running headers. Even that is not foolproof, because those headers might be suppressed, such as on even-numbered pages at chapter end.

It should have been relatively simple for TEX to provide absolute-page-position commands.

No grid typesetting The boxes-and-glue model of typesetting that TEX exploits so successfully has one drawback: if it is required that all typeset material line up on a grid, it can be extremely difficult to guarantee this in TEX. The reason is that there are a great many dimensional parameters in TEX that have a certain amount of stretch associated with them, and it isn't possible to tell TEX that it should limit spacing and stretch to multiples of a grid unit. Also, in the presence of inherently two-dimensional material, like mathematics, music, figures, and tables, it is difficult in TEX to guarantee grid-aligned results.

Even if high-quality typography would reject grid-based typesetting, there is nevertheless a demand for it for certain document types, and it needs to be available in the typesetting system.

No comments in DVI files The DVI format permits a single comment in the preamble, but has no mechanism for embedding comments anywhere else. Programmers have always found comments useful, and a few other programs besides TEX can produce DVI files. The `\special` command could, of course, be used for this purpose, but it is already rather a mess.

Some people feel that it would be useful to preserve all input in document-formatter output, so that the original text could be recovered. Consider, for example, a programmer faced with the task after a corporate merger of updating company names in all documentation, much of which exists only in DVI, PostScript, and PDF format. Specially-formatted comments would be one way for input to be hidden in the output.

No rotated material TEX provides no direct way to typeset material at some angle relative to the horizontal. This capability is sometimes required

[19] `http://ctan.tug.org/tex-archive/macros/plain/contrib/xii.tex`.

for landscape display of tables, and for labeling of axes and curves in graphs.

PostScript allows text to be typeset along arbitrary curved paths, and allows the coordinate system to be scaled, rotated, and translated. Text rotation later became possible from TEX via `\special` commands for a suitable DVI-to-PostScript driver, as described in [16, Chapter 11].

What did METAFONT do right?

METAFONT's manual [31] has many similarities with TEX's [29], including charming illustrations by Duane Bibby.

METAFONT is used directly by many fewer people than TEX is: although we all use, and sometimes generate, fonts, few of us have the interest, need, and skill to design them. Indeed, authors of books and journal articles rarely have any choice: publishers, editors, designers, and marketing staff have already made all font decisions.

The next two sections are therefore considerably shorter than the preceding pair on TEX.

Open software The most important done-right feature of METAFONT is that it is an *open-source literate program* [32], and the same favorable comments made above for TEX apply to METAFONT.

This openness made it possible for John Hobby to produce METAPOST [21] [22, Chapter 13], a substantial modification of METAFONT intended for creation of drawings, as well as fonts, with output in PostScript instead of GF and TFM files.

Font-design kernel METAFONT has a small kernel of primitive commands that are highly suited to font design.

Programming language Unlike TEX, METAFONT is a true programming language, though perhaps not as general as one might like.

One of the very interesting features of the METAFONT language is the ability to define characters in terms of constraint equations, to ensure, for example, that both legs of the letters H, M, and N have the same width. Font formats like PostScript Type 1, TrueType, and OpenType provide a feature called *hints* with a similar purpose, but they are less powerful than METAFONT equations.

Although TEX actually does use native floating-point arithmetic for some internal glue calculations that cannot affect line breaking or page breaking, METAFONT has no floating-point arithmetic whatever in either the language, or the program.

'Meta' fonts A superb achievement of Knuth's work on the Computer Modern family of typefaces [33] is that he was not only able to reproduce closely an existing traditional font (Monotype Modern 8A, used in his *Art of Computer Programming* treatise), but to generalize the character programs with a few dozen well-chosen parameters in such a way as to allow changes in those parameters to produce several different styles that are still recognizably members of the Computer Modern type family, and to allow tiny tweaks to those characters to accommodate the imaging characteristics of various output devices.

Before METAFONT, no font designer had the tools to pull off such a design coup. Since then, only Adobe's no-longer-supported Multiple Master font technology has attempted to do something similar, and even then, the results were much less flexible than METAFONT can produce, and only a few such fonts were marketed. For an Adobe insider's view of their failure, see [10].

For technical text, the lack of a broad choice of families of related fonts is a serious problem: Alan Hoenig [22, pp. 316–344] shows 29 samples of the same page of a scientific article with various font choices.

Apart from Computer Modern, there are only a few families with a repertoire that includes a typewriter font and a mathematical font: Bigelow & Holmes' Lucida is probably the best example, and there was once talk of Adobe extending the Stone family to add them. The widely-used Times family lacks Times-like sans-serif and typewriter fonts: most books set in Times use Helvetica and Courier for that purpose, but they are a poor match.

Shaped pens An interesting feature of METAFONT is that drawing of characters is done by moving a pen of a user-definable shape along a curved path: the envelope of the pen shape traces strokes of the character. The tracing can add dots to, or subtract dots from, the drawing. Also, the dots need not just be on or off: they can have small signed integer values. At output time, the positive values become black dots, and the zero or negative values, white dots.

This can be compared with character descriptions in PostScript Type 1, TrueType, and OpenType fonts, which are based on describing paths that are then stroked with a fixed-shape pen, or closed and filled with a solid color.

Open font formats The format of the GF and TFM font files produced by METAFONT is well documented in [31, Appendices F and G] and [32, §45 and §46], and the `gftype` and `tftopl` utilities in standard TEX distributions can produce human-readable dumps. The companion program `pltotf` can convert a possibly-modified property-list dump back into a TFM file.

This situation should be contrasted with the secrecy surrounding most commercial fonts before TEX: even the PostScript Type 1 font format was not documented until competition from the TrueType camp forced Adobe to publish the black-and-white book [25], and the hinting in some TrueType and OpenType fonts is encumbered by vendor patents.

What did METAFONT do wrong?

Bitmap output premature In the 1970s, typesetting technology was moving away from the 500-year tradition of hot-lead type, and the much more recent optical-mask generation of character shapes, to a digital representation of the shapes in grids of tiny dots that can be displayed on dot-matrix, ink-jet, and laser printers.

Although METAFONT character descriptions are in terms of continuously-varying pen strokes, the shape that is recorded in the GF file is just a compressed bitmap at a particular resolution. This made the job of DVI translators easier: they could either copy those little bitmaps into a large page-image bitmap, or they could encode them in a bitmap font format understood by a particular output model, such as Hewlett-Packard PCL or Adobe PostScript.

One significant effect of this decision is that the font resolution must be chosen at DVI translation time. That is acceptable when the DVI output is sent immediately to an output device with a matching resolution and imaging technology.

A few years later, PostScript appeared with a different model: fonts would normally be represented by outline shapes, and those outlines would be either resident in, or downloaded to, a PostScript interpreter in the printing device. Since that interpreter could be specially tuned for each device, it could handle the conversion of shapes to bitmaps. Since hints are embedded in the font files, rather than the font programs, they could be applied during rasterization. With initial laser-printer resolutions of about 300 dots/inch, typical characters contained only a few hundred dots in the bitmap, and that rasterization could be done acceptably fast at print time, as long as shapes, once converted to bitmaps, were cached for reuse.

The benefit of the PostScript (and later, PDF) approach is twofold: dependence on device resolution and device-imaging characteristics is moved from fonts to output devices, and character shape information is preserved, so that documents viewed under any magnification will retain smooth characters.

While PostScript Type 1 font outline representations of all Computer Modern fonts are now freely available, it has taken nearly two decades, and a lot of hard work by several groups of people, to achieve that.

Pen shapes While pen shapes are a powerful feature, their use in fonts designed with METAFONT makes it very hard to translate to other font formats that lack that feature. This can only be called a misfeature of METAFONT if one needs to cater to other font formats, but most of us have to.

For a long time, most PostScript-producing DVI drivers simply output fonts as PostScript Type 3 bitmap fonts, with the result that when PDF conversion became commonplace, screen quality was horrid.

This need not have been the case, since Adobe's own co-founder, and chief architect of PostScript, had long ago shown how to convert high-resolution character bitmaps to gray-scale displays [59], and the `xdvi` translator on Unix systems has always done a superb job of bitmap-font display.

It took ten years after the appearance of PDF for a new release of Adobe's own Acrobat Reader to improve the display of bitmap fonts, and even then, the improved version is not yet available on any Unix platform.

Fortunately, researchers at the Nottingham font conversion project have found clever ways to replace bitmap fonts in PostScript files with outline fonts [51]. Their software should allow repair of a lot of existing PostScript files, such as Web documents, for which TEX source files are unavailable. Once converted to PDF, those files should have much better screen readability.

Curve representations The computer graphics and computer-aided design community in the 1960s and 1970s developed well-understood, and widely-implemented, representations of curves as special polynomial forms known as Bézier and B-spline curves, and later, nonuniform rational B-splines (NURBs). The first two can represent conic sections, including circular arcs, only approximately, but NURBs can describe them exactly.

The interest in these special polynomials is that they are bounded by a companion polyline whose vertices can be moved around to obtain smooth, and humanly-predictable, variations in curve shapes. Ordinary polynomials lack this important design-control property: small changes in their parameters can often produce large, and surprising, changes in curve shape.

John Warnock, the PostScript architect, had learned about these curve representations in courses at Utah, and realized that they could be used to describe letter shapes, just as well as the shapes needed in aircraft, automobile, and ship design for which they were originally developed. PostScript Type 1 fonts are therefore based on cubic Bézier curve segments.

METAFONT supports Bézier curves, but also some more general curve types with curl and tension parameters that are difficult to reduce to the simpler curves. The `tex-fonts` mailing list[20] in 2003 carried extensive debates about how to handle these reductions, since there is considerable interest in automated conversion of fonts between any pair of popular formats.

The subject of this subsection is thus not really a criticism of METAFONT, but it has nevertheless proved a serious stumbling block for font-format conversion.

Inadequate I/O Like TEX, METAFONT too has inadequate I/O, but the situation is even worse. TEX can open an arbitrary filename for output, but the METAFONT language can only write to its log file, which is cluttered with lots of other material beyond programmer control.

One of the things that METAFONT can do is report the outlines that it discovers as it sweeps the pen shape around the character. Some of the work in translation of METAFONT output to PostScript Type 1 form has used that trace information, but the task is much harder than it could have been with a more powerful I/O model.

Font sizes Although METAFONT can be used to produce fonts with more than 256 characters, such as would be needed for some East Asian languages, the magic number 256 is still present in a way that suggests the format may not be well-suited to alphabets or syllabaries with more than 256 characters of arbitrary dimensions. TEX itself cannot handle fonts with more than 256 characters.

Inflexible character numbering Font programs for METAFONT contain character definitions, each beginning with a hard-coded declaration of the position of the character in the font. These positions then carry over into the output GF and TFM files. TEX and METAFONT have to agree on character numbering, so that a TEX character definition can declare, for example, that the mathematics Greek letter Ω is found in the tenth slot in the Computer Modern roman font. That numerical position is then fixed, and embedded in at least four separate files: the METAFONT program file, the two font files, and a TEX startup file. In practice, the redundancy is widespread: I found 44 different files in the CTAN archives that declared the position of Ω, and that problem is repeated for several hundred other characters in common use with TEX.

PostScript Type 1 fonts take a different, and more flexible, approach. Character definitions are given names in the `.pfa` (PostScript font ASCII) or `.pfb` (PostScript font binary) outline font file, and both names and numbered positions in the `.afm` (Adobe font metric) file. However, the number is used only for a default position.

What really determines the character position in the font is the *font encoding vector*, a list of up to 256 names that can be specified in PostScript code outside the font file itself. Only a handful of standard encoding vectors [4, Appendix E] are defined and known to all PostScript interpreters, even though there are thousands of different Type 1 fonts. Most Latin text fonts use the encoding vector named `StandardEncoding`, and therefore omit its definition from the font files.

As shown at TUG'2001 [6], about 20% of Type 1 fonts actually contain more than 256 characters, even though only 256 can be accessed from one encoding vector. However, the same font can be loaded by a PostScript program and assigned different internal names and encoding vectors, so with a little more work, all characters can still be accessed in single PostScript job.

Had METAFONT used character names instead of numbers, and provided a TEX-accessible encoding vector, many of the difficulties in using non-METAFONT fonts in TEX would disappear, and TEX virtual fonts would be rarely needed.

Not adopted by font designers Many of us expected that professional font designers would use METAFONT to create new implementations of old fonts, and entirely new fonts. This has not happened, despite Donald Knuth's extensive collaboration with noted font designers Chuck Bigelow, Richard Southall, and Hermann Zapf. Richard in particular taught us at early TEX Users Group conferences that font designers are highly-skilled artists, craftsmen, and draftsmen; they are not programmers, and describing character shapes in an abstract programming language, like METAFONT, is not an easy task.

[20] `http://www.math.utah.edu/mailman/listinfo/tex-fonts/`.

This is unfortunate, because I think that font designers could make great progress with 'meta'ness. Perhaps significant gains will come from an entirely different direction: the important work of Wai Wong and Candy Yiu presented at TUG'2003 on the programmatic representation of Chinese characters, combined with progress in optical character recognition, could make possible the scanning of tens of thousands of characters, and the automatic creation of METAFONT programs to regenerate them in a range of beautiful shapes and styles.

The huge East Asian character repertoire is the biggest hurdle for font vendors to address as the world moves to Unicode. Although there are more than 20,000 fonts with 256 or fewer characters,[21] there is hardly a handful of Unicode fonts yet.[22] None is even close to complete, and none comes in a family of styles.

Future directions

While TEX, `troff`, and commercial desktop-publishing systems can continue to be used as before, I believe quite strongly that the electronic representation of documents in the future is going to involve two key technologies:

1. XML, XML, XML, XML, XML, XML, ...
2. Unicode and ISO 10646 character encoding.

XML directions I have already given economic reasons why publishers are interested in XML. If the archived document at the publisher's end is going to be XML, perhaps authors should be writing in XML to begin with. The major problem seems to be the lack of good XML tools, and a wide selection of sample document type definitions (SGML and XML DTDs correspond roughly to LATEX class files).

Like HTML, XML [14] is a particular instance of the Standard Generalized Markup Language, SGML [15].

Because SGML is exceeding complex and general, it is very difficult to write parsers for it: two of the best-known freely-available ones are James Clark's `sgmls`[23] (22,000 lines of C) and `sp`[24] (78,000 lines of C++ and C).

XML is a reaction to this complexity: its major design goal is that it should be possible for any competent programmer to write a working parser for XML in an afternoon. XML removes much of the freedom of SGML, and eliminates character-set variations by adopting Unicode. For most applications were SGML might be used, XML is good enough.

I spent several months in 2002–2003 on a joint book project authored directly in XML (and typeset by `troff`, though TEX could have done the job too if the publisher had provided the needed tools), and found that the project has not been notably harder than it would have been with LATEX markup. Fortunately, there is virtually no mathematics in the book, because that would have been very painful to produce in XML with the current state of input tools.

The major LATEX feature that I've missed is the ability to define new commands to obtain consistent formatting of certain technical names. Once I'd prepared a powerful Emacs editing mode for XML,[25] input of the more verbose XML tags took about as few keystrokes as I would have needed with my LATEX editing mode.[26]

A significant advantage of XML is that many markup mistakes were quickly caught by a rigorous SGML parser *before* typesetting began: with LATEX, the mistake would often not be evident until the typeset output was proofread.

I missed a companion to `html-pretty` during the book work, so I ultimately wrote a workable, but still rudimentary, equivalent for XML.[27]

Unicode directions The biggest difficulty for the future is likely to be Unicode, not XML.

First, Unicode requires very much larger fonts, and the few that are currently available lack many glyphs, including virtually everything that is not a Latin, Greek, Hebrew, or Arabic relative.

Second, Unicode raises the issue of how strings of characters are to be interpreted and displayed. Those accustomed to the languages of Europe and the Americas are used to alphabets, and words displayed in the same order as they are spelled and present in the input stream. This is not always so.

Ancient inscriptions were often written in lines that alternated reading direction, a practice called *boustrophedon*, Greek for *as the ox follows the plow*.

Lao, Khmer, and Thai are based on alphabets of reasonable size (70 to 116 characters, including

[21] `http://www.math.utah.edu/~beebe/fonts/fonts-to-vendors.html`.

[22] `http://www.math.utah.edu/~beebe/fonts/unicode.html`.

[23] `http://www.math.utah.edu/pub/sgml/sgmls/`.

[24] `http://www.math.utah.edu/pub/sgml/` and `http://www.jclark.com/`.

[25] `http://www.math.utah.edu/pub/emacs/docbook.el` and `http://www.math.utah.edu/pub/emacs/docbookmenu.el`.

[26] `http://www.math.utah.edu/pub/emacs/latex.el`, `http://www.math.utah.edu/pub/emacs/ltxaccnt.el`, and `http://www.math.utah.edu/pub/emacs/ltxmenu.el`.

[27] `http://www.math.utah.edu/pub/xmlfixup/`: the name `xmlpretty` is already in use on the Internet.

letters, digits, and punctuation), and are written from left to right. However, in these languages, and several ancient ones, there are no spaces between words, only between sentences, as shown in Figure 4. Without input word-boundary marks, hyphenation and line breaking are insurmountable problems.

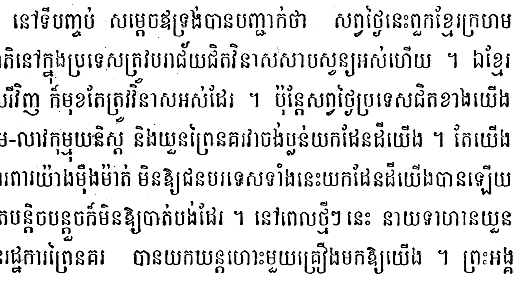

នៅទីបញ្ចប់ សម្ដេចឪទ្រង់បានបញ្ជាក់ថា សព្វថ្ងៃនេះពួកខ្មែរក្រហម
ក្បត់ជាតិនៅក្នុងប្រទេសត្រូវបរាជ័យជិតវិនាសសាបសូន្យអស់ហើយ ។ ឯខ្មែរ
ខ្សៀវសេរីវិញ ក៏មុខតែត្រូវវិនាសអស់ដែរ ។ ប៉ុន្តែសព្វថ្ងៃប្រទេសជិតខាងយើង
គឺសៀម-លាវកុម្មុយនិស្ត និងយួនព្រៃនគររាចង់ប្លន់យកដែនដីយើង ។ តែយើង
បានការពារយ៉ាងម៉ឺងម៉ាត់ មិនឱ្យជនបរទេសទាំងនេះយកដែនដីយើងបានឡើយ
សូម្បីតែបន្តិចបន្តួចក៏មិនឱ្យបាត់បង់ដែរ ។ នៅពេលថ្មីៗនេះ នាយទាហានយួន
ម្នាក់នៃរដ្ឋការព្រៃនគរ បានយកយន្តហោះមួយគ្រឿងមកឱ្យយើង ។ ព្រះអង្គ

Figure 4: Sample of Khmer text, from Franklin E. Huffman (ed.), *Intermediate Cambodian Reader*, Yale University Press, New Haven and London (1972), p. 274. There are no interword spaces. The isolated symbol that looks like a digit 7 is called *khan*; it is the Khmer end-of-sentence mark.

Hindi, and several other Indic scripts, are also based on an alphabet (about 100 characters cover letters, digits, and punctuation), but characters in Hindi are often combined into new shapes, and are often displayed in an order different from their input sequence, as shown in Figure 5.

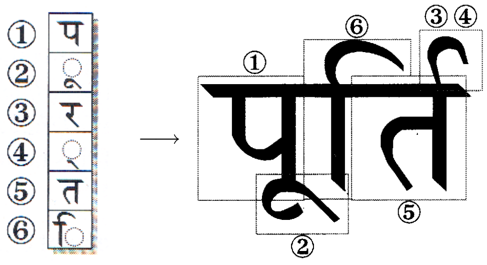


Figure 5: A Hindi word, adapted from [58, p. 17], with circled digits marking the input character order.

While Arabic is a nicely-flowing calligraphic script written from right to left along a common baseline, with spaces between words, text justification is normally done by stretching horizontal strokes in letters, instead of the spaces. Also, each Arabic letter has three shapes, depending on whether it appears first, medial, or last in the word.

Urdu uses a script closely related to Arabic, but the language is Indo-European, not Semitic, and the spoken form is often mutually comprehensible with Hindi speakers in north India and Pakistan, as shown in the trilingual dictionary entry in Figure 6.

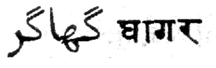
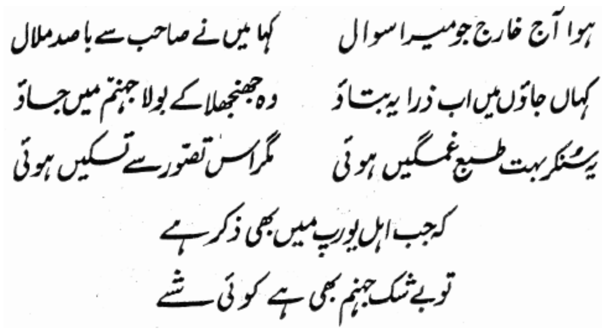

گھاگر घागर *ghāgar*, s.m. (Dahk.), The rope tied to the foot of an elefant.

Figure 6: An Urdu/Hindi/English dictionary entry.

Despite its Arabic appearance, characters in Urdu words tend to march in from the Northeast Frontier (remember, this is a right-to-left script), instead of hugging a horizontal baseline, as illustrated by the poem in Figure 7.


ہوا آج خارج جو میرا سوال — کہا میں نے صاحب سے با صد ملال
کہاں جاؤں میں اب ذرا یہ بتاؤ — وہ جھنجھلا کے بولا جہنم میں جاؤ
یہ سنکر بہت طبع غمگیں ہوئی — مگر اس تصور سے تسکیں ہوئی
کہ جب اہل یورپ میں بھی ذکر ہے
تو بے شک جہنم بھی ہے کوئی شے

Today when my petition was rejected
I asked the Sahib, feeling much dejected,
'Where shall I go to now Sir? Kindly tell.'
He growled at me and answered 'Go to Hell!'
I left him, and my heart was really sinking;
But soon I started feeling better, thinking,
'A European said so! In that case
At any rate there must *be* such a place!'

Figure 7: Urdu satirical verse of Akbar Ilahabadi. From Ralph Russell, *The Pursuit of Urdu Literature: A Select History*, Zed Books, London (1992), p. 153.

These are only a small sampling of some of the difficulties in store as the world moves to a common character-set encoding. A lot has already been accomplished, but a great deal more remains to be done. It has become clear that Ω development is not just *TEX extended for Unicode*: a typesetting system capable of handling all of the world's writing systems must be able to do much more than TEX can.

For further reading on Unicode issues, I recommend *Unicode Demystified* [13] and *Digital Typography Using LATEX* [57].

References

[1] P. H. Abbott, D. G. Brush, C. W. Clark III, C. J. Crone, J. R. Ehrman, G. W. Ewart, C. A.

Goodrich, M. Hack, J. S. Kapernick, B. J. Minchau, W. C. Shepard, R. M. Smith, Sr., R. Tallman, S. Walkowiak, A. Watanabe, and W. R. White. Architecture and software support in IBM S/390 Parallel Enterprise Servers for IEEE floating-point arithmetic. *IBM Journal of Research and Development*, 43(5/6):723–760, 1999. CODEN IBMJAE. ISSN 0018-8646. URL `http://www.research.ibm.com/journal/rd/435/abbott.html`. Besides important history of the development of the S/360 floating-point architecture, this paper has a good description of IBM's algorithm for exact decimal-to-binary conversion, complementing earlier ones [55, 9, 36, 7, 56].

[2] ACM/SIGGRAPH. Status report of the Graphic Standards Planning Committee of ACM/SIGGRAPH. *ACM SIGGRAPH—Computer Graphics*, 11(3), 1977.

[3] ACM/SIGGRAPH. Status report of the Graphic Standards Planning Committee of ACM/SIGGRAPH. *ACM SIGGRAPH—Computer Graphics*, 13(3), August 1979.

[4] Adobe Systems Incorporated. *PostScript Language Reference*. Addison-Wesley, Reading, MA, USA, third edition, 1999. ISBN 0-201-37922-8. xii + 897 pp. LCCN QA76.73.P67 P67 1999. US$49.95, CDN$74.95. URL `http://www.adobe.com/products/postscript/pdfs/PLRM.pdf`. This new edition defines PostScript Language Level 3. An electronic version of the book is available at the Adobe Web site, and is also included in a CD-ROM attached to the book.

[5] Zeev Becker and Daniel Berry. `triroff`, an adaptation of the device-independent `troff` for formatting tri-directional text. *Electronic Publishing — Origination, Dissemination, and Design*, 2(3):119–142, October 1989. CODEN EPODEU. ISSN 0894-3982.

[6] Nelson Beebe. The TEX font panel. *TUGboat*, 22 (3):220–227, September 2001. ISSN 0896-3207.

[7] Robert G. Burger and R. Kent Dybvig. Printing floating-point numbers quickly and accurately. *ACM SIGPLAN Notices*, 31(5):108–116, May 1996. CODEN SINODQ. ISSN 0362-1340. URL `http://www.acm.org:80/pubs/citations/proceedings/pldi/231379/p108-burger/`. This paper offers a significantly faster algorithm than that of [55], together with a correctness proof and an implementation in Scheme. See also [9, 1, 56].

[8] David Carlisle. A seasonal puzzle: XII. *TUGboat*, 19(4):348, December 1998. ISSN 0896-3207.

[9] William D. Clinger. How to read floating point numbers accurately. *ACM SIGPLAN Notices*, 25(6):92–101, June 1990. CODEN SINODQ. ISBN 0-89791-364-7. ISSN 0362-1340. URL `http://www.acm.org:80/pubs/citations/proceedings/pldi/93542/p92-clinger/`. See also output algorithms in [36, 55, 7, 1, 56].

[10] Stephen Coles and Thomas Phinney. Adobe & MM fonts: Insight from the inside. *Typographica*, October 9, 2003. URL `http://typographi.ca/000706.php`.

[11] W. H. J. Feijen, A. J. M. van Gasteren, D. Gries, and J. Misra, editors. *Beauty is our business: a birthday salute to Edsger W. Dijkstra*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1990. ISBN 0-387-97299-4. xix + 453 pp. LCCN QA76 .B326 1990.

[12] Jonathan Fine. Instant Preview and the TEX daemon. *TUGboat*, 22(4):292–298, December 2001. ISSN 0896-3207.

[13] Richard Gillam. *Unicode demystified: a practical programmer's guide to the encoding standard*. Addison-Wesley, Reading, MA, USA, 2003. ISBN 0-201-70052-2. xxxiii + 853 pp. LCCN QA76.6 .G5535 2002. UK£37.99.

[14] Charles F. Goldfarb and Paul Prescod. *The XML Handbook*. Prentice-Hall PTR, Upper Saddle River, NJ 07458, USA, 1998. ISBN 0-13-081152-1. xliv + 639 pp. LCCN QA76.76.H92G65 1998. US$44.95. URL `http://www.phptr.com/ptrbooks/ptr_0130811521.html`.

[15] Charles F. Goldfarb and Yuri Rubinsky. *The SGML handbook*. Clarendon Press, Oxford, UK, 1990. ISBN 0-19-853737-9. xxiv + 663 pp. LCCN Z286.E43 G64 1990. US$75.00.

[16] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LATEX Companion*. Tools and Techniques for Computer Typesetting. Addison-Wesley, Reading, MA, USA, second edition, 1994. ISBN 0-201-54199-8. xxi + 530 pp. LCCN Z253.4.L38 G66 1994. US$34.25.

[17] Michel Goossens and Sebastian Rahtz. *The LATEX Web companion: integrating TEX, HTML, and XML*. Tools and Techniques for Computer Typesetting. Addison-Wesley Longman, Harlow, Essex CM20 2JE, England, 1999. ISBN 0-201-43311-7. xxii + 522 pp. LCCN QA76.76.H94G66 1999. US$36.95. With Eitan M. Gurari and Ross Moore and Robert S. Sutor.

[18] Michel Goossens, Sebastian Rahtz, and Frank Mittelbach. *The LATEX Graphics Companion: Illustrating Documents with TEX and PostScript*. Tools and Techniques for Computer Typesetting. Addison-Wesley, Reading, MA, USA, 1997. ISBN 0-201-85469-4. xxi + 554 pp. LCCN Z253.4.L38G663 1997. US$39.75.

[19] Andrew Marc Greene. BASIX: An interpreter written in TEX. *TUGboat*, 11(3):381–392, September 1990. ISSN 0896-3207.

[20] David Gries. Binary to decimal, one more time. In Feijen et al. [11], chapter 16, pages 141–148. ISBN 0-387-97299-4. LCCN QA76 .B326 1990. This paper presents an alternate proof of Knuth's algorithm [36] for conversion between decimal and fixed-point binary numbers.

[21] John D. Hobby. A METAFONT-like system with PostScript output. *TUGboat*, 10(4):505–512, December 1989. ISSN 0896-3207.

[22] Alan Hoenig. *TEX Unbound: LATEX and TEX Strategies for Fonts, Graphics, & More*. Oxford University Press, Walton Street, Oxford OX2 6DP, UK, 1998. ISBN 0-19-509686-X (paperback), 0-19-509685-1 (hardcover). ix + 580 pp. LCCN Z253.4.L38H64 1997. US$60.00 (hardcover), US$35.00 (paperback). URL `http://www.oup-usa.org/gcdocs/gc_0195096851.html`.

[23] Allen I. Holub. *Compiler Design in C*. Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1990. ISBN 0-13-155045-4. xviii + 924 pp. LCCN QA76.76.C65 H65 1990. Prentice-Hall Software Series, Editor: Brian W. Kernighan.

[24] IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. IEEE, New York, NY, USA, August 12, 1985. ISBN 1-55937-653-8. 20 pp. US$35.00. URL `http://standards.ieee.org/reading/ieee/std_public/description/busarch/754-1985_desc.html`. Revised 1990. A preliminary draft was published in the January 1980 issue of IEEE Computer, together with several companion articles. Also standardized as *IEC 60559 (1989-01) Binary floating-point arithmetic for microprocessor systems*.

[25] Adobe Systems Incorporated. *Adobe Type 1 Font Format—Version 1.1*. Addison-Wesley, Reading, MA, USA, August 1990. ISBN 0-201-57044-0. iii + 103 pp. LCCN QA76.73.P67 A36 1990. US$14.95. URL `http://partners.adobe.com/asn/developer/pdfs/tn/T1_SPEC.PDF`.

[26] Donald Knuth. Virtual Fonts: More Fun for Grand Wizards. *TUGboat*, 11(1):13–23, April 1990. ISSN 0896-3207.

[27] Donald Knuth and Pierre MacKay. Mixing right-to-left texts with left-to-right texts. *TUGboat*, 8(1): 14–25, April 1987. ISSN 0896-3207.

[28] Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6): 607–639, December 1965. CODEN IFCNA4. ISSN 0019-9958. Russian translation by A. A. Muchnik in *Îazyki i Avtomaty*, ed. by A. N. Maslov and É. D. Stotskiĭ (Moscow: Mir, 1975), 9–42. Reprinted in *Great Papers in Computer Science* (1996) [43].

[29] Donald E. Knuth. *The TEXbook*, volume A of *Computers and Typesetting*. Addison-Wesley, Reading, MA, USA, 1986. ISBN 0-201-13447-0. ix + 483 pp. LCCN Z253.4.T47 K58 1986.

[30] Donald E. Knuth. *TEX: The Program*, volume B of *Computers and Typesetting*. Addison-Wesley, Reading, MA, USA, 1986. ISBN 0-201-13437-3. xv + 594 pp. LCCN Z253.4.T47 K578 1986.

[31] Donald E. Knuth. *The METAFONTbook*, volume C of *Computers and Typesetting*. Addison-Wesley, Reading, MA, USA, 1986. ISBN 0-201-13445-4. xi + 361 pp. LCCN Z250.8.M46 K58 1986.

[32] Donald E. Knuth. *METAFONT: The Program*, volume D of *Computers and Typesetting*. Addison-Wesley, Reading, MA, USA, 1986. ISBN 0-201-13438-1. xv + 560 pp. LCCN Z250.8.M46 K578 1986.

[33] Donald E. Knuth. *Computer Modern Typefaces*, volume E of *Computers and Typesetting*. Addison-Wesley, Reading, MA, USA, 1986. ISBN 0-201-13446-2. xv + 588 pp. LCCN Z250.8.M46 K574 1986.

[34] Donald E. Knuth. The errors of TEX. Technical Report STAN-CS-88-1223, Stanford University, Department of Computer Science, September 1988. See [35].

[35] Donald E. Knuth. The errors of TEX. *Software — Practice and Experience*, 19(7):607–685, July 1989. CODEN SPEXBL. ISSN 0038-0644. This is an updated version of [34]. Reprinted with additions and corrections in [37, pp. 243–339].

[36] Donald E. Knuth. A simple program whose proof isn't. In Feijen et al. [11], chapter 27, pages 233–242. ISBN 0-387-97299-4. LCCN QA76 .B326 1990. This paper discusses the algorithm used in TEX for converting between decimal and scaled fixed-point binary values, and for guaranteeing a minimum number of digits in the decimal representation. See also [9] for decimal to binary conversion, [55, 56] for binary to decimal conversion, and [20] for an alternate proof of Knuth's algorithm.

[37] Donald E. Knuth. *Literate Programming*. CSLI Lecture Notes Number 27. Stanford University Center for the Study of Language and Information, Stanford, CA, USA, 1992. ISBN 0-937073-80-6 (paper), 0-937073-81-4 (cloth). xv + 368 pp. LCCN QA76.6.K644. US$24.95.

[38] Donald E. Knuth. *Digital Typography*. CSLI Publications, Stanford, CA, USA, 1999. ISBN 1-57586-011-2 (cloth), 1-57586-010-4 (paperback). xvi + 685 pp. LCCN Z249.3.K59 1998. US$90.00 (cloth), US$39.95 (paperback).

[39] Donald E. Knuth and Silvio Levy. *The CWEB System of Structured Documentation, Version 3.0*. Addison-Wesley, Reading, MA, USA, 1993. ISBN 0-201-57569-8. 226 pp. LCCN QA76.9.D3 K6 1993.

[40] Donald E. Knuth and Michael F. Plass. Breaking paragraphs into lines. *Software — Practice and Experience*, 11(11):1119–1184, November 1981. CODEN SPEXBL. ISSN 0038-0644.

[41] Leslie Lamport. *LATEX—A Document Preparation System—User's Guide and Reference Manual*. Addison-Wesley, Reading, MA, USA, 1985. ISBN 0-201-15790-X. xiv + 242 pp. LCCN Z253.4.L38 L35 1986.

[42] Leslie Lamport. *LATEX: A Document Preparation System: User's Guide and Reference Manual*. Addison-Wesley, Reading, MA, USA, second edition, 1994. ISBN 0-201-52983-1. xvi + 272 pp. LCCN

Z253.4.L38L35 1994. Reprinted with corrections in 1996.

[43] Phillip Laplante, editor. *Great papers in computer science*. IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1996. ISBN 0-314-06365-X (paperback), 0-07-031112-4 (hardcover). iv + 717 pp. LCCN QA76 .G686 1996. US$23.95. URL `http://bit.csc.lsu.edu/~chen/GreatPapers.html`.

[44] Ken Lunde. *Understanding Japanese Information Processing*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 1993. ISBN 1-56592-043-0. xxxii + 435 pp. LCCN PL524.5.L86 1993. US$29.95.

[45] Ken Lunde. *CJKV Information Processing: Chinese, Japanese, Korean & Vietnamese Computing*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 1999. ISBN 1-56592-224-7. 1174 pp. LCCN PL1074.5 .L85 1999. US$64.95. URL `http://www.oreilly.com/catalog/cjkvinfo/`.

[46] Frank Mittelbach. An environment for multicolumn output. *TUGboat*, 10(3):407–415, November 1989. ISSN 0896-3207.

[47] Sao Khai Mong. A Fortran version of METAFONT. *TUGboat*, 3(2):25, October 1982. ISSN 0896-3207.

[48] Peter G. Neumann. *Computer-Related Risks*. Addison-Wesley, Reading, MA, USA, 1995. ISBN 0-201-55805-X. xv + 367 pp. LCCN QA76.5 .N424 1995. URL `http://www.csl.sri.com/neumann.html`.

[49] Arthur Ogawa. REVTEX version 4.0, an authoring package by the American Physical Society. *TUGboat*, 22(3):131–133, September 2001. ISSN 0896-3207.

[50] Michael F. Plass and Donald E. Knuth. Choosing better line breaks. In J. Nievergelt, G. Coray, J.-D. Nicoud, and A. C. Shaw, editors, *Document Preparation Systems: A Collection of Survey Articles*, pages 221–242. Elsevier North-Holland, Inc., New York, NY, USA, 1982. ISBN 0-444-86493-8. LCCN Z244 .D63 1982. US$46.50.

[51] S. G. Probets and D. F. Brailsford. Substituting outline fonts for bitmap fonts in archived PDF files. *Software — Practice and Experience*, 33(9):885–899, July 25, 2003. CODEN SPEXBL. ISSN 0038-0644. URL `http://www.eprg.org/research/`.

[52] Tomas Rokicki. Packed (PK) font file format. *TUGboat*, 6(3):115–120, November 1985. ISSN 0896-3207.

[53] Luigi Semenzato and Edward Wang. A text processing language should be first a programming language. *TUGboat*, 12(3):434–441, November 1991. ISSN 0896-3207.

[54] E. Wayne Sewell. *Weaving a Program: Literate Programming in* `WEB`. Van Nostrand Reinhold, New York, NY, USA, 1989. ISBN 0-442-31946-0. xx + 556 pp. LCCN QA76.73.W24 S491 1989.

[55] Guy L. Steele Jr. and Jon L. White. How to print floating-point numbers accurately. *ACM SIGPLAN Notices*, 25(6):112–126, June 1990. CODEN SINODQ. ISSN 0362-1340. See also input algorithm in [9], and a faster output algorithm in [7] and [36], IBM S/360 algorithms in [1] for both IEEE 754 and S/360 formats, and a twenty-year retrospective [56]. In electronic mail dated Wed, 27 Jun 1990 11:55:36 EDT, Guy Steele reported that an intrepid pre-SIGPLAN 90 conference implementation of what is stated in the paper revealed 3 mistakes:

1. Table 5 (page 124):
   insert `k <-- 0` after assertion, and also delete `k <-- 0` from Table 6.
2. Table 9 (page 125):

for	`-1:USER!("");`
substitute	`-1:USER!("0");`

   and delete the comment.
3. Table 10 (page 125):

for	`fill(-k, "0")`
substitute	`fill(-k-1, "0")`

[56] Guy L. Steele Jr. and Jon L. White. How to print floating-point numbers accurately. In ACM, editor, *20 Years of the ACM/SIGPLAN Conference on Programming Language Design and Implementation (1979–1999): A Selection*. ACM Press, New York, NY 10036, USA, 2003. ISBN 1-58113-623-4.

[57] Apostolos Syropoulos, Antonis Tsolomitis, and Nick Sofroniou. *Digital typography using LATEX*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 2003. ISBN 0-387-95217-9. xxix + 510 pp. LCCN Z253.4.L38 S97 2003.

[58] The Unicode Consortium. *The Unicode Standard, Version 4.0*. Addison-Wesley, Reading, MA, USA, 2003. ISBN 0-321-18578-1. xxxviii + 1462 pp. LCCN QA268 .U545 2004. URL `http://www.unicode.org/versions/Unicode4.0.0/`. Includes CD-ROM.

[59] J. E. Warnock. The display of characters using gray level sample arrays. *Computer Graphics*, 14 (3):302–307, July 1980. CODEN CGRADI. ISSN 0097-8930.

[60] Stefan Wohlfeil. *On the Pagination of Complex, Book-Like Documents*. Shaker Verlag, Aachen and Maastricht, The Netherlands, 1998. ISBN 3-8265-3304-6. 224 pp. DM 98.00. URL `http://www.shaker.de/Online-Gesamtkatalog/Details.idc?ID=24201&CC=59311&IDSRC=1&ISBN=3-8265-3304-6&Reihe=15`.