Chapter 10

The Errors of TEX (1989)

[Software creation involves much more than the writing of programs. The present chapter describes the milieu of literate programming, by tracing the history of all changes made to *TEX* as that system evolved. Knuth was asked by the editors of Software—Practice & Experience to prepare an article discussing the development of *TEX*; his response is reprinted here from the July 1989 issue of that journal.]

Introduction

I make mistakes. I always have, and I probably always will. But I like to think that I learn something, every time I go astray. In fact, one of my favorite poems consists of the following lines by Piet Hein **[7]:**

> The road to wisdom? Well, it's plain
> and simple to express:
> Err
> and err
> and err again
> but less
> and less
> and less.

The date today, **as** I begin to write this paper, is May **5,** 1987, exactly ten years since I began to work intensively on software systems for typesetting. I have certainly learned a lot during those ten years, judging from the number of mistakes I made; and I would like to share what I've learned with other people who are developing software. The best way to do this, as far as I know, is to present a list of all the errors that were corrected in TEX while it was being developed, and to attempt to analyze those errors.

When I mentioned my plan for this paper to Paul M. B. Vitányi, he told me about a best-selling book that his grand-uncle had written for civil engineers, devoted entirely to descriptions of foundation work that had proved to be defective. The preface to that book [25] says

> It is natural that engineers should not wish to draw attention to their mistakes, but failures are sometimes due to causes of which there has been no previous experience or of which no information is available. An engineer cannot be blamed for not foreseeing the unknown, and in such cases his reputation would not be harmed if full details of the design and of the phenomena that caused the failure were published for the guidance of others. ... To be forewarned is to be forearmed.

In my own case I cannot claim that "unknown" factors lay behind my blunders, since I was totally in control of my programming environment. I can justly be blamed for every mistake I made, and I'm certainly not proud of the record. But I see no harm in admitting the horrible truth about my tendency to err, when such details might shed light on the problem of writing large programs. (Besides, I'm lucky enough to have a secure job.)

Empirical studies of programming errors, conducted by Endres [5] and by Basili and Perricone [1], have already led to interesting results and to the conclusion that "more data must be collected on different projects." I can't claim that the data presented below will be as generally applicable as theirs, because all of the programming I shall discuss was done by one person (me). Insightful models of truly large-scale software development and program evolution have been introduced by Belady and Lehman [3]. However, I do have one advantage that the authors of previous studies did not have; namely, the entire program for TEX has been published [23]. Hence I can give fairly precise information about the type and location of each error. The concept of scale cannot easily be communicated by means of numeric data alone; I believe that a detailed list gives important insights that cannot be gained from statistical summaries.

Types of Error

Some people undoubtedly think that everything I did on TEX was an error, from start to finish. But I shall consider only a limited class of errors here, based on the log books I kept while I was developing the .program. Whenever I made a change, I noted it down for future reference, and it is these changes that I shall discuss in detail. Edited forms of my log books are appended below [Chapter 11].

I guess I could say that this paper is about 'changes', not 'errors', because many of the changes were made in order to introduce new features rather than to correct malfunctions. However, new features are necessary only when a design is deficient (or at least non-optimal). Hence, I'll continue to say that each change represents an error, even though I know that no complex system will ever be error-free in this extended sense.

The errors in my log books have each been assigned to one of fifteen general categories for purposes of analysis:

- **A,** an algorithm awry. Here my original method proved to be incorrect or inadequate, so I needed to change the procedure. For example, error #212 fixed a problem in which footnotes appeared on a page backwards: The last footnote came out first.

- B, a blunder or botch. Here I knew what I ought to do, but I wrote something else that was syntactically correct—sort of a mental typo. For example, in error #26 I wrote 'before' when I meant 'after' and vice versa. I was thinking so much of the Big Picture that I didn't have enough brainpower left to get the small details right.

- C, a cleanup for consistency or clarity. Here I changed the rules of the language to make things easier to remember and/or more logical. Sometimes this was just a surface change to TEX's "syntactic sugar," as in #16 where I decided that **\input** would be a better name than **\require.**

- D, a data-structure debacle. Here I didn't properly update the representation of information to preserve the appropriate invariants. For example, in error #105 I failed to return nodes to available memory when they were no longer accessible.

- E, an efficiency enhancement. Here I changed the program so that it would run faster; the existing code was correct but slow. For example, in error #287 I decided to give TEX the ability to preload font information, since it took awhile to read thirty short files at the beginning of every run.

- **F,** a forgotten function. Here I didn't remember to do everything I had intended, when I actually got around to writing a particular part of the code. It was a simple error of omission, rather than commission. For example, in error #11 and again in #172 I had a loop of the form '**while** p $\neq$ null **do**', and I forgot to advance the pointer p inside the loop! This seems to be one of my favorite mistakes: I often forget the most obvious things.

• *G*, a generalization or growth of ability. Here I realized that some extension of the existing specifications was desirable. For example, error #303 generalized my original primitive command '`\ifT` {char)', which tested if a given character was 'T' or not, to the primitive '\if ⟨char⟩⟨char⟩', which tested if two given characters were equal. Eventually, in **#666**, I decided to generalize further and allow '\if (token)(token)'.

• I, an interactive improvement. Here I made TEX respond better to the user's needs. Sometimes I saw how to help TEX identify and recover from errors in the documents it was processing. I also kept searching for better ways to communicate the reasons underlying TEX's behavior, by making diagnostic information available in symbolic form. For example, error #54 introduced '...' into the display of context lines so that users could easily tell when information was truncated.

• L, a language liability. Here I misused or misunderstood the programming language or system hardware I was working with. For example, in error #24 I wanted to reduce a counter modulo 8, so I wrote '$t := (t-1) \bmod 8$'; this unfortunately made t negative because of the way 'mod' was defined. Sometimes I forgot the precedence of operators, etc.

• M, a mismatch between modules. Here I forgot the conventions I had built into a subroutine when I actually got around to using that subroutine. For example, in error #64 I had a macro with four parameters (x_0, y_0, x_1, y_1) that define a rectangle; but when I used it, I gave the parameters in different order, (x_0, x_1, y_0, y_1). Such "interface errors" included cases when a procedure had unwanted side effects (like clobbering a global variable) that I failed to take into account. Some mismatches (like incorrect data types) were caught by the compiler and not entered in my log.

• P, a promotion of portability. Here I changed the organization or documentation of the program; this affected only a person who would try to read or modify the code, not a person who tried to run it. For example, in error **#59**, one of my comments about how to set the size of memory had '$\ge$' where I meant to say '$\le$'. (Most changes of this kind were not recorded in my log; I noted only the noteworthy ones.)

• Q, a quest for quality. Here 1 changed the specifications of what the program should output from given input, when I learned how to improve the typographic appearance of the output. For example,

error **#187** changed TEX's behavior when typesetting formulas that have an unusually complex superscript; as a result, TEX now produces

$$e^{\frac{1}{1-q_j^2}} \qquad \text{instead of} \qquad e_{\frac{1}{1-q_j^2}}\,.$$

• R, a reinforcement of robustness. Whenever I realized that TEX could loop or crash in the presence of certain erroneous input, I tried to make the code bulletproof. For example, error **#200** made sure that a user-supplied character number was between 0 and 127; otherwise parts of TEX's memory could be wiped out.

• S, a surprising scenario. Errors of type S were particularly bad bugs that forced me to change my original ideas, because of unforeseen interactions between various parts of the program. For example, error #25 was logged when I first discovered a consequence of TEX's convention about blank lines denoting the end of a paragraph: There's often a blank space in TEX's internal data structure just before a paragraph ends, because a space is usually supplied at the end of the line just preceding a blank line. Thus I had to write new code to delete the unwanted space. Whenever such unexpected phenomena showed up, I had to go back to the drawing board and fix the design.

• T, a trivial typo. Sometimes I didn't type the right thing when I entered the program into the computer, although my original pencil draft was correct. For example, in error #48 I had typed '−' instead of '+'. If a typing mistake was detected by the compiler as a syntax error, I didn't log it, because bad syntax can easily be corrected.

Nine of these categories (**A**, B, D, F, L, M, R, S, T) represent "bugs"; such errors absolutely had to be corrected. The other six categories (C, E, G, I, P, Q) represent "enhancements"; I could have refused to consider the existing situation erroneous. As remarked earlier, I'm considering all items in the log to be indications of error. But there is a significant difference between errors of these two kinds: I felt guilty when fixing the bugs, but I felt virtuous when making the enhancements.

My classification of errors into fifteen categories is ad hoc, but at the moment it's the best way I can think of to make sense out of my experiences. Some of the bug categories refer to simple flaws in the basic mechanics of programming: Writing the right thing but typing it wrong (T); thinking the right thing but writing it wrong (B); knowing the right thing but forgetting to think it (F); imperfectly knowing the tools (L) or the specifications (M). Such bugs are easy to fix once they've been identified. Categories A and D represent the next level of difficulty, as we get into technical aspects of what programming is all about. (As Niklaus Wirth has said, Algorithms + Data Structures = Programs.) Category R covers the special situation in which we want a program to survive even when its input is incorrect. Finally, category S accounts for higher-level surprises; these are the subtle bugs that result from complex interactions between different parts of a system. Thus the nine types of bugs have a somewhat logical structure. The remaining six categories—cleanliness (C), efficiency (E), generalization (G), interaction (I), portability (P), and quality (Q)—seem to provide a reasonable way to classify the various kinds of enhancements that were made to TEX during its development.

My classification scheme relies more on essential functionality than on the external form of the program. Thus it isn't easy to use my statistics about the number of errors per category to answer questions like "How many bugs were due to improper use of **goto** statements?" Such questions are interesting to teachers of programming, but I no longer think that they are extremely important. If I had indexed my errors by syntactic categories, I would have found that errors #45, #91, #119, #155, #231, #352, #354, #419, #523, #581, and #801 could be ascribed to my use or abuse of **goto**; also #512 could be added to this list, since return and **goto** are analogous. Thus we can conclude from my experience with TEX that **goto** statements can indeed be harmful. On the other hand we must balance this fact with the realization that bad gotos account for only 1.4% of my errors; we must identify other culprits if we're going to do away with the other 98.6%. Sure enough, several other errors were caused by lapses in my use of other control structures: A case statement got me in trouble in #21; a while confused me in #29; if-then-else led me astray in #467, #471, #680, and #843. (See also #796 and #845, where efficiency of control was important.) I conclude that every feature of a programming language can be harmful, if it is misused.

Some of the errors noted in my log book were much more devastating than others. In certain cases the changes were far-reaching, affecting dozens of different parts of the program; several days of "hacking" were necessary before such changes had been made and verified. For example, change #110 required major surgery to the program, because my original ideas were incapable of handling aligned tables inside of aligned tables. On the other hand, some of my errors were only venial sins, and some of the changes were merely twiddles; for example, #87 simply improved the wording of a diagnostic message. Although the log doesn't give an explicit weighting to the errors, the "heavy" errors tend to cancel with

the "light" ones, so we can still get a reasonable insight into the stability of the program if we calculate, say, the number of errors logged per year.

Chronology

The development of TEX has taken place over a period of ten years, and the lessons I learned can best be understood when they're put into the context of the other things I was doing during that time. Typography has many facets, hence TEX itself was only one of the projects I decided to work on. The two most significant companion systems were METAFONT (a system for typeface design) and Computer Modern (a family of typefaces defined in terms of the METAFONT language); these programs had to be debugged just as TEX did, and their debugging logs show a similar development history. I also needed a dozen or so utility routines to support TEX and METAFONT; the most notable of these are TANGLE and WEAVE, which constitute the WEB system of structured documentation [20,18].

Beginnings

The genesis of TEX probably took place on February 1, 1977, when I first chanced to see the output of a high-resolution typesetting machine. I was told that this fine typography (the galley proofs of a book by Winston [26], which our faculty was considering for inclusion on an exam syllabus) was produced by entirely digital methods; yet I could see no difference between the digital type and "real" type. Therefore I realized that a central aspect of printing had been reduced to bit manipulation. As a computer scientist, I couldn't resist the challenge of improving print quality by manipulating those bits better. Therefore my diary entry for February 8 says that, already at that time, I began discussing the possibility of new typesetting software with people at Stanford's Artificial Intelligence Lab. By February 13 I had changed my plan to spend a forthcoming sabbatical year in South America; instead of traveling to an exotic place and working on Volume 4 of The Art of Computer Programming, I had decided to stay at Stanford and work on digital typography.

I mentioned earlier that the design of TEX was begun on May 5, 1977. A week later, I wrote a draft report containing what I thought was a pretty complete design, and I stayed up until 5 a.m. typing it into the computer. The problem of typesetting seemed quite straightforward, so I soon started thinking about fonts instead; I spent the next 45 days writing a program that was destined to evolve into METAFONT. By June 28, I had 25 lowercase letters in various styles that looked reasonably good to me at the time; and three days later I figured out how to handle the 26th letter, which required some new ideas [15].

I went back to thinking about TEX on July 3. Several people had made thoughtful comments on my earlier draft, and I prepared a thoroughly revised language definition after two weeks of further study. (This included two days of working with dictionaries in order to develop an algorithm for hyphenation of English.) The resulting document, I thought, was a reasonably complete specification of a language for typesetting, and I left it in the capable hands of two graduate students who were my research assistants that summer (Frank Liang and Michael Plass). Their job was to implement TEX while I flew off for a visit to China. I returned on August 25 and had just one day to meet with them before leaving on another three-week trip. On September 14 I returned and they presented me with a sheet of paper that had been typeset by their proto-TEX program! They had implemented only about 15% of the language, and they had used data structures that were not general enough or efficient enough to support the remaining 85%; but they had chosen their subset wisely so that a small test program could run from start to finish. Hence it was easy for me to imagine what a complete system would entail.

Now it was time for Liang and Plass to go back to school, and time for my sabbatical year to begin. I started coding the "final version of TEX" (or so I thought) on September 16, and immediately I discovered that their summer work represented a truly heroic achievement. Although I had thought that my specification of TEX was quite complete, I encountered loose ends every 15 minutes or so when I was actually faced with writing the code. I soon realized that if I had been in my students' shoes—having to implement this language when the author was completely unreachable—I would have thrown up my hands in despair; important policy decisions had to be made at every turn.

That was the first big lesson I learned during my work with TEX: The designer of a new kind of system must participate fully in the implementation. Even if I had been available for consultation with my students, they would have had to come to me so often with questions that the work would have dragged on forever. I can imagine them having to spend a half hour or so explaining each particular problem to me, and we would have needed literally hundreds of those meetings. Now I knew why other projects I'd heard about, in which the language designer had decided not be the compiler writer, had failed.

By October 14 I had coded all of TEX except for the parts that typeset mathematics, and except for the routines that convert from TEX's internal representation into codes for an output device. At this point I

had to leave for three weeks of travel in Europe. This European trip had been planned long before, so it was mostly unrelated to typesetting; but 1 did have some interesting discussions about curve-drawing with mathematicians I met in Oberwolfach, Germany, and in Oslo, Norway. I also was able to arrange a visit to the headquarters of Monotype Corporation in Redhill, England.

After returning I spent November finishing the numerals, uppercase letters, and punctuation marks of the first-draft Computer Modern types. I needed to have a complete, font because I had been invited to give a lecture about this work to the American Mathematical Society, and I didn't want to have only lowercase examples to show. I prepared the AMS lecture [12] during December and presented it in January, so I didn't have a chance to resume the coding of TEX until January 14. But finally I was able to write the following in my diary on February 10, 1978:

> Finished the TEX programs including all loose ends and got them all compiled without syntax errors (4 a.m.).

TEX was the first fairly large program I had written since 1970; so it was my first nontrivial "structured program," in the sense that I wrote it while consciously applying the methodology I had learned in the early 70s from Dijkstra, Hoare, Dahl, and others. I found that structured programming greatly increased my confidence in the correctness of the code, while the code still existed only on paper. Therefore I could wait until the whole program was written, before trying to debug any of it. This saved lots of time, because I didn't have to prepare "dummy" versions of nonexistent modules while testing modules that were already written; I could test everything in its final environment. Of course I had a few qualms in January about whether my code from September would really work; but that gave me more of an incentive to finish the whole thing sooner.

Even on February 10, when TEX had been compiled and was ready to be tested, I didn't feel any compelling need to try it immediately. I knew that the program was fairly readable and "informally proved correct," so I spent the next month making italic, greek, script, symbols, and large delimiter fonts. My test program for TEX required those fonts, so I didn't want to start testing until everything was in place. Again, I knew I was saving time by not having to prepare prototypes that would merely simulate the real thing; structured programming gave me the courage to wait until the whole system was ready. I finished the large symbols on March 8, and I happily penned the following in my diary on March 9:

> Entered all accumulated corrections to TEX program and compiled it—tomorrow the debugging begins!

My log book for errors in TEX began that next day, March 10; the debugging process will be discussed below. By March 29 I had decided that TEX was essentially working,

> ... (except perhaps for error recovery)—it's time to celebrate!

I began tuning up the fonts and drafting ideas for a user manual; then I spent a few days at Alphatype Corporation in Illinois, from whom Stanford had decided to purchase a phototypesetter. From April 11 to May 11, I took time off from typography to work on dozens of updates to Seminumerical Algorithms, which is Volume 2 of The Art of Computer Programming [10]; I wanted to incorporate new research results into that text, which was to be TEX's first big application. Then on May 14 I began to get TEX running again; proof copies of pages iv through 8 of Volume 2 came out of our Xerox Graphics Printer on May 15.

My work was cut out for me during the next weeks: I became a production user of TEX, typing the manuscript of Volume 2. This proved to be an invaluable experience, as explained below. By the time my sabbatical year ended, on September 24, I had finished the typing up to page 441 of that 700-page book. Improvements to TEX kept occurring to me all during that time, of course—except during a month-long vacation trip with my family. (Even on vacation I kept seeing fonts everywhere and thinking about how to draw such letterforms by computer. I spent one morning sitting by one of the trails in the Grand Canyon designing the algebraic notation for METAFONT; my fonts had previously been written in a primitive macro language and compiled directly into machine code, not interpreted.) I also spent three weeks that summer writing the first manual for TEX.

Although my sabbatical year was over, I kept working on typography in odd moments between classes in the fall; the text of Volume 2 was completed on the morning of November 15. On November 17 I began writing METAFONT, and my diary entry for December 31, 1978 was this:

> Finished the METAFONT interpreter, just in time to celebrate New Year's eve (11:59 p.m.).

Other people had begun to use TEX in August of 1978, and I was surprised to see how fast the system was propagating. I spent my spare time during the first three months of 1979 thinking about how to make TEX available in Pascal form. (The original program was written in SAIL, a language that was available on only a few computers.) During this

period I began to experiment with the typesetting of Pascal programs; I wrote a program called BLAISE that converted Pascal source code into a TEX file for pretty-printing. BLAISE soon developed into a system called DOC for structured documentation, completed on March 31, 1979; programs in DOC format could be converted either to Pascal or to TEX. Luis Trabb Pardo and Ignacio Zabala subsequently used DOC to prepare a highly portable version of TEX in Pascal, completed in April of 1980.

About this time I learned another big lesson: Writing software is much harder than writing books. I couldn't simultaneously teach classes well and finish what needed to be done on typography. So I asked to be excused from teaching in the spring of 1979; my diary for March 22 said,

> Now my obligations are fairly well cleared away and it's back to the stalled research on TEX.

(It turned out that I was able to teach during only 13 of the 21 academic quarters between my sabbatical years in that period. I continued to supervise graduate students, but I gave no classroom lectures during 1983 when the work on TEX and METAFONT was at its peak; I also missed three months in 82, 84, and 85. I really enjoy teaching, but I couldn't see any way to finish the TEX project without relinquishing almost all of my other duties.)

On April 1, 1979, I returned to METAFONT, which had been written but not debugged. METAFONT began to work on April 28. Then I began to design software for the Alphatype machine; that took about three months. During the summer I wrote the METAFONT manual, which gave me further experience with TEX. And TEX also received an important stimulus from the American Mathematical Society that summer, when several people (including Barbara Beeton and Michael Spivak) were given the opportunity to spend some time at Stanford developing TEX macros. The AMS people introduced me to several important applications, such as the indexes to Mathematical Reviews, which stretched TEX to its limits and led to substantial improvements.

Endings

By August 14, 1979, I felt that TEX was essentially complete and fairly stable. I lectured that evening to about 100 participants of the Western Institute for Computer Science in Santa Cruz, telling about my experiences developing and debugging the program. At that time my log book of errors had accumulated 420 items; little did I know that the final total would be more than twice that! But already I knew that I had learned a lot by keeping the log, and I must have been enthusiastic because I lectured from 7:30 to 9:30 p.m. (The audience was equally enthusiastic—they kept asking me questions until 11:30 p.m. So I resolved to write a paper about the errors of TEX, and at last I am able to do so.)

I devoted the last months of 1979 and the first months of 1980 to Computer Modern, which needed to be rewritten in terms of the new METRFONT. Then I needed to update Volume 2 again—computer science marches inexorably forward—until I finally had finished producing camera-ready copy on our Alphatype. This was the goal I had hoped to achieve during my sabbatical year; I reached it at 2 a.m. on July 29, 1980, about two years late. During the rest of 1980 I wrote papers about what 1 thought were the most novel ideas in TEX [16] and in METRFONT [17].

But my research on TEX was by no means finished. About 50 people from all over the USA met at Stanford on February 22, 1980, and established the TEX Users Group (TUG). I asked them if they would mind my cleaning up the language in several upward-incompatible ways, even though this would make the user manual and their existing computer files obsolete; and nobody objected to such changes! Soon TUG grew dramatically, under the able chairmanship of Richard Palais, and it became international. I realized that I could not disappoint all these people by leaving TEX in its current state and returning immediately to work on subsequent volumes of The Art of Computer Programming.

I needed to work out a better "endgame strategy," and it soon became clear what ought to be done: The original versions of TEX and METAFONT should be scrapped, once they had served their purpose of accumulating enough user experience to indicate what such languages ought to be. New versions of TEX and METRFONT should be written, designed to last a long time and to be highly portable between computers and typesetting devices of all kinds. Moreover, these new programs should be published, because TEX was making it possible to improve the state of the art of program documentation. I decided to do my best to produce a stable system and to explain all I knew about it, so that other people could take it over and maintain it if it proved to be important. This way I could return to other pursuits in good conscience, knowing that if my typographic research had any merit it would be carried on by others in whatever ways would prove to be necessary.

So that was my new goal; I thought I could achieve it in one or two more years. The original TEX program was renamed TEX78, and the new one was to be called TEX82.

Classes and miscellaneous chores kept me too busy to do much else during the first half of 1981, but I began to write TEX82 on August 22. By September 9 I realized that the DOC system needed to be completely revised, so I spent two months replacing it by a much better system called WEB [18]. Since then my programming language of choice has been WEB (which, unlike DOC, was written in its own language). After a month in Europe, I was able to resume writing TEX82 on December 1, 1981. The draft of TEX82 was completed on June 29, 1982; as before, I wrote the entire program before trying to run any of it.

Meanwhile I had other problems to worry about. When my new copy of Seminumerical Algorithms arrived in January of 1981, I had expected to be filled with joy at the consummation of so much hard work. Instead, I burned with disappointment, as I realized that I still had a great deal to learn about fonts. The early Computer Modern typefaces were not at all what I had hoped to achieve, when I first saw them in print. They had looked reasonably good at low resolution, so I had blithely assumed that high resolution would be much better. Not so. My education in typefaces was barely beginning. I met Richard Southall later in 1981, a professor of type design who had exactly the expertise I was lacking; so I invited him to visit Stanford. We spent the entire month of April, 1982, working about 16 hours a day, revising Computer Modern from A to z.

I debugged TEX82 in the summer of '82, then began to write the new manual—called The *TEXbook* [21]—in October. The first manual had been written hastily and finished in 21 days, but I wanted The *TEXbook* to meet much higher standards. Therefore I wasn't able to finish it until a full year later.

It was during this period, October '82 to October '83, that TEX became a mature system. I had to rethink every aspect of its design as I rewrote the manual. Fortunately I was aided by a wonderful group of knowledgeable volunteers, who would meet with me for two or three hours every Friday noon and we would discuss the tradeoffs of every important decision. The diverse backgrounds of these people provided an important counterweight to my one-sided views. Finally, on December 9, 1983, I decided that the first phase of my endgame strategy was complete; I gratefully hosted a coming-of-age party for TEX, with 36 guests of honor, at the Fuki-Sushi restaurant in Palo Alto.

The rest is history. I wrote METAFONT in WEB between December, 1983 and July, 1984; I wrote The *METAFONTbook* between August, 1984, and October, 1985, taking off five months (February through July) to rewrite Computer Modern in terms of the new METAFONT. I began another sabbatical year in October, 1985, just after the TEX project disbanded. Finally, after adding a few more finishing touches, I was able to celebrate the long-planned completion of my "endgame" on May 21, 1986, when my publishers sponsored a reception at the Computer Museum in Boston; that was the day I first saw the five hardcover volumes of Computers & Typesetting, the books that summarized my nine years of work on TEX, METAFONT, and Computer Modern.

Another year has gone by and I would like to report that TEX has proved to be 100% correct. But I cannot, not yet. For I stumbled across a hidden TEX anomaly last January. And I've just been, teaching a course about software development based on the internal structure of TEX; students in the class have noticed a few things that should be improved. So I suppose there is still at least one bug lurking there. I plan to hold off publishing this paper until another year or so has gone by, so that I'll have more reason to believe that my log book of errors is complete.

Contents of the Log Books

As I said, an appendix to this paper [Chapter 11] reproduces the entire list of errors that I kept as TEX was changing. The best way to comprehend how TEX evolved is to peruse this list. The first 519 items refer to the original program TEX78, which was written in SAIL, from the time I began to debug it to the time I stopped maintaining it. The remaining items, numbered 520–849 (as of May 1987), refer to the "real" program TEX82, which was written in WEB. I didn't keep any record of errors removed during the hectic period when TEX82 was being debugged,[1] but items 520 and following include every change that was made to TEX82 after it passed its first test. The differences between TEX78 and TEX82, seen from a user's standpoint, have been listed elsewhere [2].

I've tried to edit the log entries so that they can be understood in terms of the published listing [23] of TEX82. For example,

15 Add the forgotten case 'set-font:' to *eq_destroy*. §275 F

(Footnote added July 1991) When I wrote the above I had forgotten about another log book that I had in fact kept during those hectic days. A complete list of the changes made while I debugged my first large-scale "literate program" has just turned up in the Stanford University Archives. I am inserting those entries into Chapter 11, so that the record is now complete. The newly discovered entries are numbered X1–X343.

is entry #15. My original log entry actually referred to case '[font]' in 'eqdestroy' using SAIL syntax, but I've changed to Pascal syntax in the edited log. Similarly, the 1978 identifier font eventually became set-font, so I've adopted the published equivalent. TEX82 contains a procedure called eq-destroy in §275 of the program, and this procedure is quite similar to the eqdestroy of TEX78; so I've supplied §275 as a program reference. (It turns out that eq-destroy no longer needs a 'set-font:' subcase, but it did in 1978.) The 'F' after §275 means that this was a bug of type F, a forgotten function.

Changes to a program often spawn other changes later. I've tried to indicate that phenomenon in the appendix by prefixing the number of a prior error when it was an important part of the reason for a subsequent error. Thus #67 is

25 ↦ 67 Replace the space at paragraph end by fillglue, not by zero. §816 B

Error #25 was logged when I had been surprised to find a space at the end of TEX's internal representation of a paragraph. I had "cured" the problem by converting the space from a normal interword space to a space of width zero. But that wasn't good enough, since it was possible for TEX to try breaking a line at the zero-width space. A better solution was to replace the space by the glue that is always added to fill out the end of a paragraph.

Figure 1 shows a time chart of the first 519 log entries—the errors of TEX78. There's a burst of activity right near the beginning, since I logged the first 237 errors during the three weeks of initial debugging. Thus the main line in Figure 1, which shows the cumulative number of errors as a function of time, is nearly horizontal at the beginning. But it's nearly vertical at the end, since only 13 changes were made during the last year of TEX78's activity.

Another line also appears in Figure 1: It represents the total number of different pages I typeset with TEX78 as I was experimenting with the first version. The dotted line in July 1978 stands for the 200 pages of the first TEX manual, and the dotted line in June 1979 stands for the 100 pages of the first METAFONT manual; the remaining solid lines stand for the 700 pages of Volume 2 and some experiments with DOC.

Figure 1 shows that four different phases can be distinguished in the development of TEX78. First came the debugging phase (Phase 0), already mentioned. Then came a longer period of time (Phase 1) when I typeset several hundred pages of Volume 2 and the first user manual; this experience suggested many amendments to my original design.

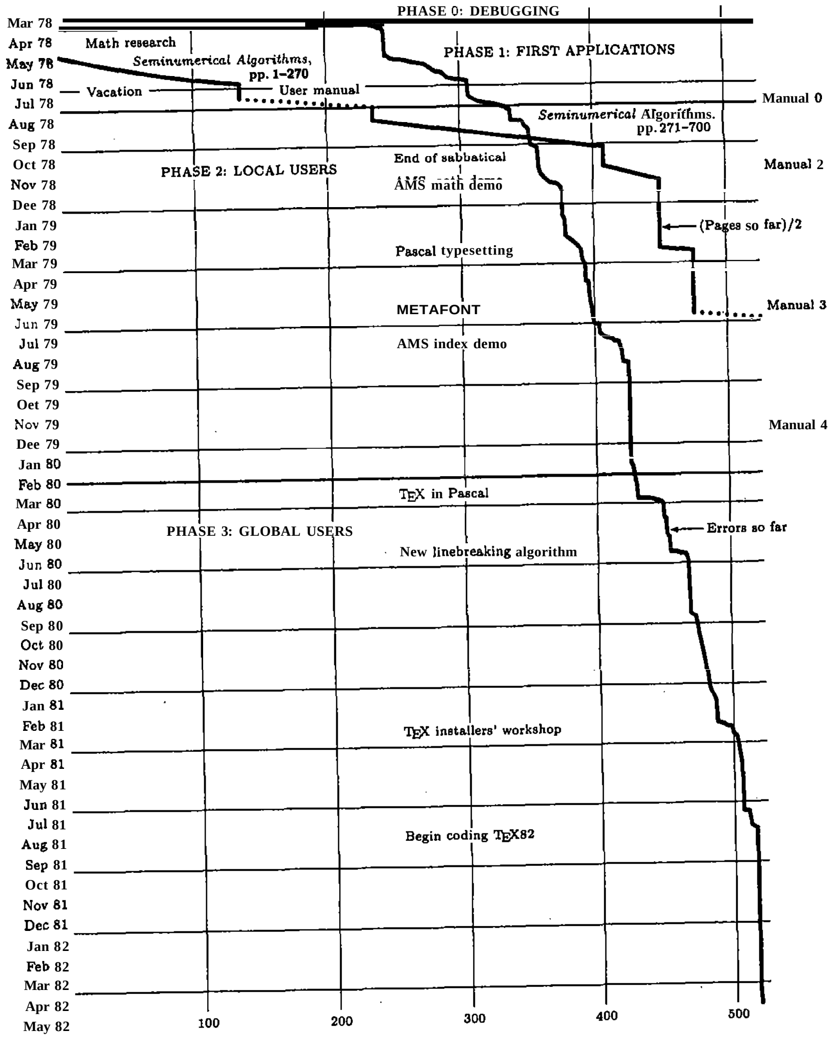

Figure 1. The rise and fall of TEX78.

errors began to show up. New users find new bugs. This coming-out phase (Phase 2) included small bursts of changes when I faced new applications—a suite of difficult test cases posed by the American Mathematical Society, then the application to Pascal formatting, then the complex index to Math Reviews. Finally there was Phase **3,** when changes were made in anticipation of a future TEX82; I wanted several new ideas to be well tested before I programmed the "ultimate" TEX.

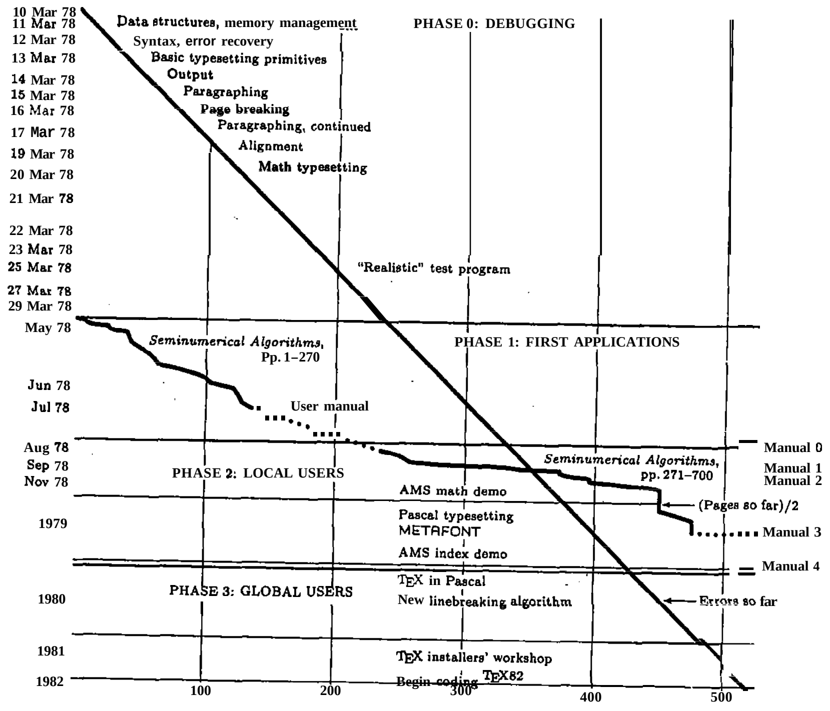

Figure 2. The errors of TEX78.

The Initial Debugging Stage

Let's roll the clock back now and look more closely at the earliest days of TEX78. In some ways this was the most interesting time, because the whole concept of TEX was just beginning to take shape. Figure 2 is a modified version of Figure 1, redrawn with a time warp. There's now exactly one error per time unit, so the 18-day debugging phase has been slowed down to almost half of the total development time; on the other hand, the years 1981–1982 at the bottom go by so fast as to be barely visible.

I mentioned that TEX78 was entirely coded before I first tried to run it on March 10. My debugging strategy was to walk through the program using the **BAIL** debugger, a system program by John Reiser that allowed me to execute the statements of my program one at a time; **BAIL** would also interpret additional SAIL statements that I entered online. Whenever I came to a section of program that I'd seen before, I could set a breakpoint and continue at high speed until coming to new material. Watching the program execute itself in this "dynamic order" has always been insightful for me, after I've desk-checked it in the "static order" of my original code.

Figure 2 shows that I got through the program initialization the first day; then I was gradually able to check out the routines for basic data management, parsing, and error reporting. On the fourth day TEX began to combine boxes and glue, and there was visible output on the fifth day. During the following three days I tested the algorithms for breaking paragraphs into lines and breaking lines into pages. All this went rather smoothly; I logged 101 errors during this first week, but all of the problems were comparatively minor oversights, to be expected in any program of this size.

On the ninth day I tackled alignment of tables, and got a big shock: My original algorithms were quite wrong. I had greatly misunderstood this aspect of TEX, because I'd greatly underestimated the complications of nested alignments. (The log mentions some of the puzzlement and frustration I felt at the time.) I wrestled with alignment for two days before finding a solution.

Then I looked at the last remaining part of TEX, the code for typesetting mathematics; this took another four days. (Well, the "days" were nights actually; I worked during the night to avoid delays due to time-sharing.) Finally I had seen essentially all of TEX in operation, and I could let it run at full speed instead of relying on single-step mode. I spent six more days helping TEX get through its first test data; finally the test was passed. Whew! The debugging phase was over, 18 days and 237 log-book entries after it began.

I kept track of how long this process took, so that I'd be better able to estimate the duration of future programming projects. Here are the figures:

Day	Time *(hours)*	Day	Time (hours)
10 Mar 1978	6	19 Mar 1978	7.5
11 Mar 1978	7	20 Mar 1978	10
12 Mar 1978	8	21 Mar 1978	8
13 Mar 1978	7	22 Mar 1978	6
14 Mar 1978	8	23 Mar 1978	7.5
15 Mar 1978	8	25 Mar 1978	7
16 Mar 1978	7	26 Mar 1978	6
17 Mar 1978	7	27 Mar 1978	8
18 Mar 1978	8	29 Mar 1978	6

The total debugging time, 132 hours, was extremely encouraging to me, because it was much less than the 41 days it had taken me to write the program. Previously I had needed to devote about 70% of program

development time to debugging, but now the figure had dropped to about 30%. I considered this to be a tremendous victory for structured programming, since my programming time had also decreased from what it had been with old habits. Later, with the WEB system, I noticed even further gains in productivity.

How big was TEX at the time? I estimated this by counting the number of semicolons (4857) and the number of occurrences of the SAIL reserved words comment (480) and else (223). Since I always put semicolons before end, the total number of statements in the program could be computed as

$$;\ -\ \text{comment} + \text{else} = 4857 - 480 + 223 = 4600.$$

Thus the debugging strategy I used allowed me to verify about 35 statements per hour.

The fact that I made 237 log entries in 132 hours means that I was logging things only about once every 33 minutes; thus the total time needed to keep the log was negligible. I can definitely recommend the practice to everybody. During most of the debugging time I was clicking away at the keys of my terminal, getting to know exactly what TEX was doing; I needed only a few extra minutes to make the log entries, which helped me get to know myself.

Early Typesetting Experience

Now that TEX was able to typeset its test program, I could proceed to my main goal, the typesetting of Volume 2. This was a somewhat tedious task—the keyboarding of a 700-page book is not one of life's greatest pleasures—but the regular appearance of nice-looking pages kept me happy. The jagged line in Figure 2 shows my progress in terms of pages typeset versus errors in the TEX log; a similar (even more jagged) line appears in Figure 1, showing pages typeset as a function of time.

The most striking thing about the jagged line in Figure 2 is that it's almost straight. Ideas about how to improve TEX kept occurring to me quite regularly as I typed the manuscript. Between May 13 and June 22 I processed about 250 pages, and added 69 new entries to the log. Those 69 entries included 29 "bugs" and 40 "enhancements"; thus, I thought of a new way to improve TEX at a regular rate of about one enhancement for every six pages typed.

I mentioned earlier my firm conviction that I could not have correctly delegated the coding of TEX to another person; I had to be doing it myself, because writing a new sort of program implies continually revising the specifications. Similarly, I could not have correctly delegated these initial typing experiments to another person. I had to put myself in the rôle of a regular user; there's no substitute for such experience, when a new system is being designed.

But at the time I wasn't thinking about creating a system that would be used widely; I was designing TEX primarily for my own use. The idea that TEX could or should be generalized to other applications besides The Art of Computer Programming dawned on me only gradually, as people kept noticing what I was doing and expressing an interest in it.

John McCarthy observed during this period that TEX was doing a reasonable job with respect to traditional mathematical copy, but he suspected that I'd have a tough time typesetting a book about TEX itself. "That will be the real test," he said, "because you'll have to shut off many of TEX's automatic features in order to handle problems of self-reference."

In July I succumbed to John's challenge and prepared a user manual for TEX. Sure enough, this experience helped me identify quite a few weaknesses in the existing design, things that I probably wouldn't have noticed if I had confined my attention to The Art of Computer Programming alone. Again I thought of enhancements at the rate of about one for every six or seven pages, as I wrote the manual; but these weren't really occasioned by defects in TEX's ability to be self-referential, as John had predicted. The new enhancements came about because the process of manual-writing forced me to think about TEX as a whole, in a new way. The perspective of a teacher/expositor helped me to notice several inconsistencies and shortcomings.

Thus, 1 came to the conclusion that the designer of a new system must not only be the implementor and the first large-scale user; the designer should also write the first user manual. The separation of any of these four components would have hurt TEX significantly. If I had not participated fully in all these activities, literally hundreds of improvements would never have been made, because I would never have thought of them or perceived why they were important.

Phases 2 and 3: Users

But a system cannot be successful if it is too strongly influenced by a single person. Once the initial design is complete and fairly robust, the real test begins as people with many different viewpoints undertake their own experiments. At the beginning of August, I distributed 45 copies of the draft manual to people who had expressed interest in using TEX and who had promised to give me feedback before the "real" user manual would be issued in September. So TEX had a multitude of users for the

first time, and I began to learn about a wide variety of new applications and perceptions.

I continued to typeset the remaining 450 pages of Volume 2, and my personal experiences with those pages continued to suggest regular improvements to TEX until I got up to about page 500. But the final 200 pages were just drudgework, not really inspirational to me in any way as far as TEX was concerned. Nor did I learn much more, except about page layout, when I typed the METAFONT manual some months later. The really important influences on TEX after the first manual was published were the users, first because they made different kinds of mistakes than I had anticipated, and later because they had important suggestions about how to improve TEX's capabilities.

Guy Steele was visiting Stanford that summer; he took a copy of TEX back to MIT with him, and I began to get feedback from two coasts. One of Guy's suggestions, which I staunchly resisted at the time, was to include some sort of mini-programming language in TEX so that users could do numerical calculations. Slowly but surely I began to understand the need for such features, which eventually became a basic part of TEX82. Another early user was Terry Winograd, who pushed TEX's early macro capabilities to their limits. He and Michael Spivak, who began to work with TEX in the summer of 1979, taught me a lot about the peculiar properties of macro expansion. Researchers at Xerox PARC also had a significant influence on TEX at this time; Lyle Ramshaw modified the program to work with Xerox's new fonts and new output devices, while Leo Guibas and Doug Wyatt undertook to rewrite TEX in the MESA language.

Figures 1 and 2 indicate that the first TEX user manual was issued in five versions. "Manual 0" was the preliminary draft, handed out to 45 guinea pigs who agreed to help me test the very first system. "Manual 1" was a Stanford technical report issued a month later; it was reprinted as "Manual 2" in November, using the higher-resolution printing devices at Xerox PARC. The American Mathematical Society published a paperback version [13] of Manual 2 in the summer of 1979; that was "Manual 3." Then Digital Press published "Manual 4," which included the METAFONT manual and some background information, in December of 1979 [14].

The publishers of manuals 3 and 4 asked readers to mail a reply card if they were interested in forming a TEX Users Group, and more than 100 people answered Yes. So the first TUG meeting, in February 1980, marked the beginning of yet another phase in the life of the SAIL program TEX78. A great influx of new users and new applications made me strive for a more complete language. Hence there was a flurry of activity at the end of March, 1980, when I decided to extend TeX in more than a dozen ways. These extensions represented only a fraction of the ideas that had been suggested, but they seemed to provide all the requested functionality in a clean way. The time was ripe to make the extensions now or never, because the first versions of TeX in Pascal were due to be released in April.

The last significant batch of changes to TeX78 were made in the summer of 1980, when TeX acquired the ability to typeset paragraphs with arbitrary shapes. Still, the error log shows that I kept adding enhancements regularly as the worldwide use of TeX continued to grow. It turned out that the final bugs corrected in TeX78 were all introduced by recent enhancements; they were not present in the program of 1978.

The most significant pattern to be found among the enhancements made to TeX78 after its earliest days is the "unbundling" of things that used to be frozen inside the code. At first I had fairly rigid ideas about how much space to put in certain places, about how much penalty to charge for certain line breaks, about how to interpret various characters in the input, and even about where to find certain characters in fonts. One by one, starting already at change #104, these things became parameters that could be changed by users who had different requirements and/or different preferences.

The Real TeX

I had vastly underestimated the complexities and subtleties of typesetting when I'd naïvely expected to work out a complete system during a single sabbatical year. But once I began, it became clear by 1980 that I had acquired almost a moral obligation to advance the art and science of typography in a more substantial way. I realized that I could never be happy with the monster I had created unless I started over and built an entirely new system, using the experience I had gained from TeX78.

I began writing the new system in the summer of 1981, and I decided to call it TeX82 because I knew it would take a year to complete. Once again I couldn't delegate the job to an associate; I wanted to rethink every detail of TeX, and I wanted to have a thorough taste of "literate programming" before I dared to inflict such ideas on others [20]. I wanted to produce truly portable software that would have a chance to serve for many years as a reliable component of larger systems. I wanted TeX82 to justify the confidence that people were placing in TeX78, which was getting more praise than it deserved.

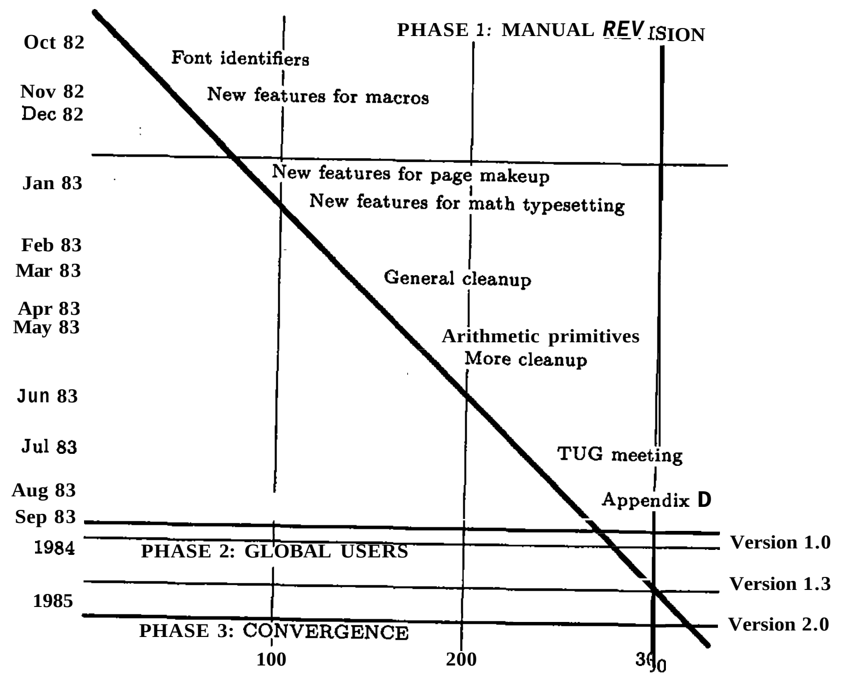

Figure 3. The errors of TeX82.

Figure 3 shows the development of TeX82, starting at the moment I decided that it was essentially bug-free; this illustration uses the same time-warp strategy as Figure 2. From the beginning there were hundreds of users, so TeX82's Phase 1 was analogous to TeX78's Phase 2. But now there was yet a new dimension: Several dozen people were also reading the code and making well-informed comments on how to improve it. Furthermore I had regular meetings with volunteer helpers who represented many different points of view. So I had a golden opportunity to hone the ideas to a new state of perfection.

Two major changes were installed very early in TeX82's history. One was to the way fonts are selected in a document (change #545), and the other was to the treatment of conditional parts of macros (change #564). Both of these changes impinged on many of the fundamental assumptions I had made when writing the code; these were definitely the most traumatic moments in TeX's medical history. I was glad to see that WEB's documentation facilities helped greatly to make such drastic revisions possible.

Phase 1 of TEX82 ended about a year after it began, when I completed writing The *TEXbook*. The log reveals that most of the changes made to TEX during 1983 relate to the chapters of the manual that I was writing at the time. This was the period when TEX really grew up. As I said above, manual writing provides an ideal incentive for system improvements, because you discover and remove glitches that you can't justify in print. When you're writing a user manual, you also have your last chance to make any enhancements that you've thought about before; if certain enhancements aren't made then, you know that you will forever wish you'd taken time to add them.

As with TEX78, the error log of enhancements to TEX82 shows a significant trend toward greater user control. More and more things that were originally hardwired in the system became parametric instead of automatic.

Phase 2 of TEX82 began with the paperback publication of The *TEXbook* and ended with the publication of the hardcover edition. During this phase (which lasted from October 1983 to May 1986) I was mostly working on METAFONT and Computer Modern, so TEX changed primarily in ways that would blend better with those systems. The log entries of Phase 2, #790 to #840, also show that a number of ever-more subtle bugs were detected by ever-more sophisticated users during this time. There was also a completely unsubtle bug, #808, which somehow had snuck through all my tests and caused no apparent harm for an amazingly long time.

Now TEX82 is in its third and final phase. It has grown from the original 4600 statements in SAIL to 1376 modules in WEB, representing about 14,000 statements in Pascal. Five volumes describing the complete systems for TEX, METAFONT, and Computer Modern have been published. No more changes will be made except to correct any bugs that still might lurk in the code (or perhaps to improve the efficiency or portability, when it's easy to do so while correcting a real bug). I hope TEX82 will remain stable at least until I finish Volume 7 of The Art of Computer Programming.

Test Programs

Since 1960 I have had extremely good luck with a method of testing that may deserve to be better known: Instead of using a normal, large application to test a software system, I generally get best results by writing a test program that no sane user would ever think of writing. My test programs are intended to break the system, to push it to its

extreme limits, to pile complication on complication, in ways that the system programmer never consciously anticipated. To prepare such test data, I get into the meanest, nastiest frame of mind that I can manage, and I write the cruelest code I can think of; then I turn around and embed that in even nastier constructions that are almost obscene. The resulting test program is so crazy that I couldn't possibly explain to anybody else what it is supposed to do; nobody else would care! But such a program proves to be an admirable way to flush the bugs out of software.

In one of my early experiments, I wrote a small compiler for Burroughs Corporation, using an interpretive language specially devised for the occasion. I rigged the interpreter so that it would count how often each instruction was interpreted; then I tested the new system by compiling a large user application. To my surprise, this big test case didn't really test much; it left more than half of the frequency counts sitting at zero! Most of my code could have been completely messed up, yet this application would have worked fine. So I wrote a nasty, artificially contrived program as described above, and of course I detected numerous new bugs while doing so. Still, I discovered that 10% of the code had not been exercised by the new test. I looked at the remaining zeros and said, Shucks, my source code wasn't nasty enough, it overlooked some special cases I had forgotten about. It was easy to add a few more statements, until eventually I had constructed a test routine that invoked all but one of the instructions in the compiler. (And I proved that the remaining instruction would never be executed in any circumstances, so I took it out.)

I used such "torture tests" to debug three compilers during the 60s. In each case very few bugs were ever discovered after the tests had been passed, so the methodology was quite effective. But when I debugged TEX78, my test program was quite tame by comparison—except when I was first testing the mathematics routines (March 20–23). I guess I wasn't trying as hard as usual to make TEX a bulletproof system, because I was still thinking of myself as TEX's main user. My original test program for TEX78 was written with an "I hope it works" attitude, rather than "I bet I can make it fail." I suppose I would have found several dozen of the bugs that showed up later (like #240 and #263) if I had stuck to the torture-test methodology. Still, considering my mood at the time, I suppose it was a good idea to have a test program that would look like real typography; I didn't know what TEX should do until I could judge the aesthetic quality of its output.

At any rate, my first test program was based on a sampling of material from Volume 2. I went through that book and boiled it down to five pages that illustrated just about every kind of typographic difficulty to be found in the entire volume. (The output of this test program can be seen in another paper [6], where David Fuchs and I used the same test data to study some algorithms for font management.)

Years later, when TEX82 was ready to be debugged, I understood pretty clearly what the program was supposed to do, so I could then apply the superior torture-test methodology. My test program was called TRIP; I spent about five days preparing the first draft of TRIP in July, 1982. Here, for example, is a relatively mild portion of the original TRIP code:

```
\def\gobble#1{} \floatingpenalty 100
\everypar{A\insert200{\baselineskip400pt\splittopskip
  \count15pt\hbox{\vadjust{\penalty999}}\hbox to-10pt{}}%
  \showthe\pagetotal\showthe\pagegoal\advance\count15by1
  \mark{\the\count15}\splitmaxdepth-1pt
  \paR\gobble} % abort every paragraph abruptly
\def\weird#1{\csname\expandafter\gobble\string#1
  \string\csname\endcsname}
\message{\the\output\weird\one}
```

(Please don't ask me what it means.) Since then I've probably spent at least 200 hours modifying and maintaining TRIP, but I consider that time well spent, and I think TRIP is one of the most significant products of the TEX project [19]. The reason is that the TRIP test has detected extremely subtle bugs in hundreds of implementations of TEX, bugs that would have been almost impossible to track down in any other way. TEX82, with its TRIP test, has proved to be much more reliable than any of the Pascal compilers it has been compiled with. In fact, I believe it's fair to say that TEX82 has helped to flush out at least one previously unknown compiler bug whenever it has been ported to a new machine or tried on a compiler that has not seen TEX before! These compiler errors were detectable because of the TRIP test. Later I developed a similar test program for METAFONT, called TRAP [22], and it too has helped to exorcise dozens of compiler bugs.

A single test program cannot detect all possible mistakes. For example, TEX might terminate with a "fatal error" in several ways, only one of which can happen on any particular run. Furthermore, TRIP runs almost automatically, so it does not test all of TEX's capability for

online interaction. But TRIP does exercise almost all of TEX's code, and it does so in tricky combinations that tend to fail if any part of TEX is damaged. Therefore it has proved to be a great time-saver: Whenever I modify TEX, I simply check that the results of the TRIP test have changed appropriately.

The only difficulty with the TRIP methodology is that I must check the output myself to see if it's correct. Sometimes I need to spend several hours before I've determined the appropriate output; and I'm fallible. So TEX might give the wrong answer without my being aware of it. This happened in bugs #543 and #722, when I learned to my surprise that TEX had never before done the correct thing with TRIP. A system utility for comparing files suffices now to convince me that incremental changes to TEX or TRIP cause the correct incremental changes to the TRIP test output; but when I began debugging, I needed to verify by hand that thousands of lines of output were accurate.

I should mention that I also believe in the merit of formal and informal correctness proofs. I generally try to prove my programs correct, informally, by stating appropriate invariants in my documentation and checking at my desk that those relations are preserved. But I can make mistakes in proofs and in specifying the conditions for correctness, just as I make mistakes in programming; therefore I don't rely entirely on correctness proofs, nor do I rely entirely on empirical test routines like TRIP.

Location and Type of Errors

Let me review again the fifteen classes of errors that are listed in my error log:

A — Algorithm	F — Forgotten	P — Portability
B — Blunder	G — Generalization	Q — Quality
C — Cleanup	I — Interaction	R — Robustness
D — Data	L — Language	S — Surprise
E — Efficiency	M — Mismatch	T — Typo

I mentioned before that each of the errors listed in the appendix refers where possible to its approximate location in the program listing of TEX82. It's natural to wonder whether the errors are uniformly interspersed throughout the code, or if certain parts were particularly vulnerable. Figure 4 shows the actual distribution. No part of the program has come through unscathed — or, shall we rather say, unimproved — but some parts have seen significantly more action. The boxes to the left of

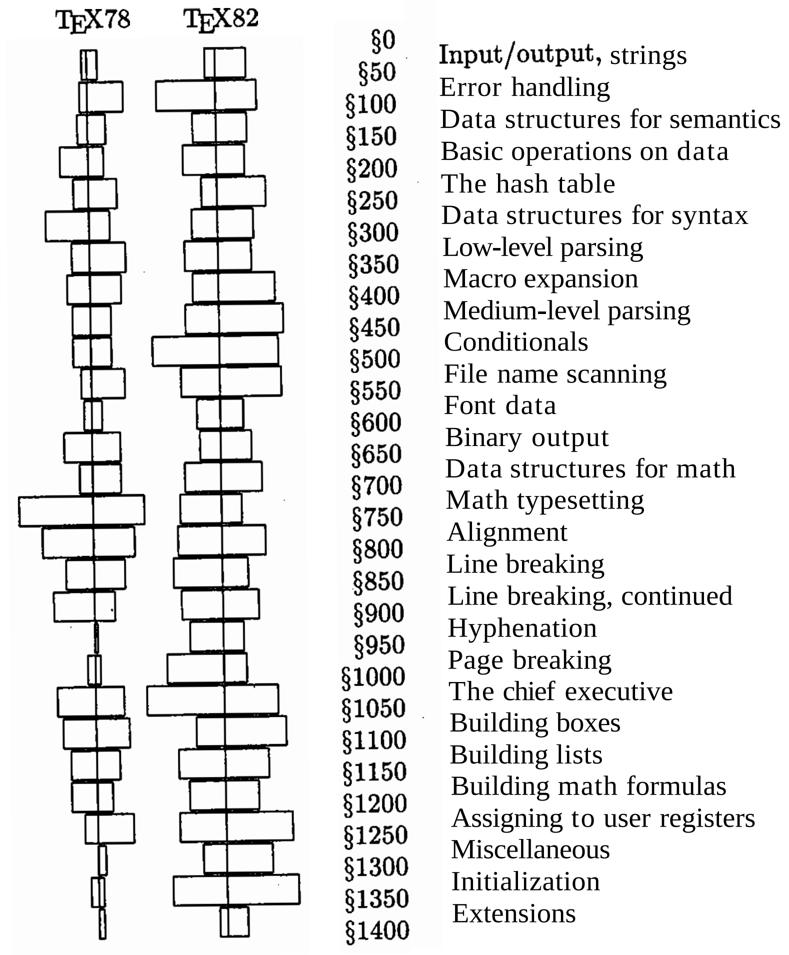

Figure 4. Distribution of [bugs|enhancements] by program location.

the vertical lines in Figure 4 represent "bugs" (A, B, D, F, L, M, R, S, T), while the boxes to the right represent "enhancements" (C, E, G, I, P, Q). The most unstable parts of TEX78 were the parts I understood least when I began to write the code, namely mathematical formatting and alignment. The most unstable parts of TEX82 were the parts that differed most from TEX78 (the conditional instructions and other aspects of macro expansion; also the increased user access to registers and internal quantities used in TEX's decision-making).

I should mention why hyphenation is almost never mentioned in the log of TEX78. Although I said earlier that TEX78 was entirely written before any of it was tested, that's not quite true. The hyphenation algorithm was quite independent of everything else and easily isolated from the code, so I had written and debugged it separately during three days in October, 1977. (There's obviously no advantage to testing independent programs simultaneously; that leads only to confusion. But the rest of TEX was highly interdependent, and it could not easily be run when any of the parts were absent, except for the routines that produced the final output.) The hyphenation algorithm of TEX78 was English-specific; Frank Liang, who had helped me with this part of TEX78, developed a much better approach in his thesis [11], and I ultimately incorporated his algorithm in TEX82 [see Chapter 8].

Figure 5 shows the accumulated number of errors of each type in TEX78, with bugs at the bottom and enhancements at the top. Initially the log entries are mostly bugs, with occasional enhancements of type I; at the end, however, enhancements C, G, and Q predominate. Figure 6 is a similar diagram for TEX82. The pattern is much the same; evidently an additional four years of experience did not teach me to make fewer mistakes.

Some Noteworthy Bugs

The gestalt of TEX's evolution can best be perceived by scanning through the log book, item by item. But I would like to single out several errors that were particularly instructive or otherwise memorable.

A, Algorithmic Anomalies.

I decided from the beginning that the algorithms of TEX would be in the public domain. But if I were to change my mind and charge a fee for my services in inventing them, I would probably request the highest price for a comparatively innocuous-looking group of statements now found in $851 and $854 of the program. This precise sequence of logical tests, used to control when a line break is being forced because there is no "feasible" alternative, has the essential form

```
if $\alpha_1 \vee \alpha_2$ then
    if $\alpha_3 \wedge \alpha_4 \wedge \alpha_5 \wedge \alpha_6$ then $\sigma_1$
    else if $\alpha_7$ then $\sigma_2$ else $\sigma_3$
else $\sigma_4$
```

and most of the appropriate boolean conditions α_i were discovered only

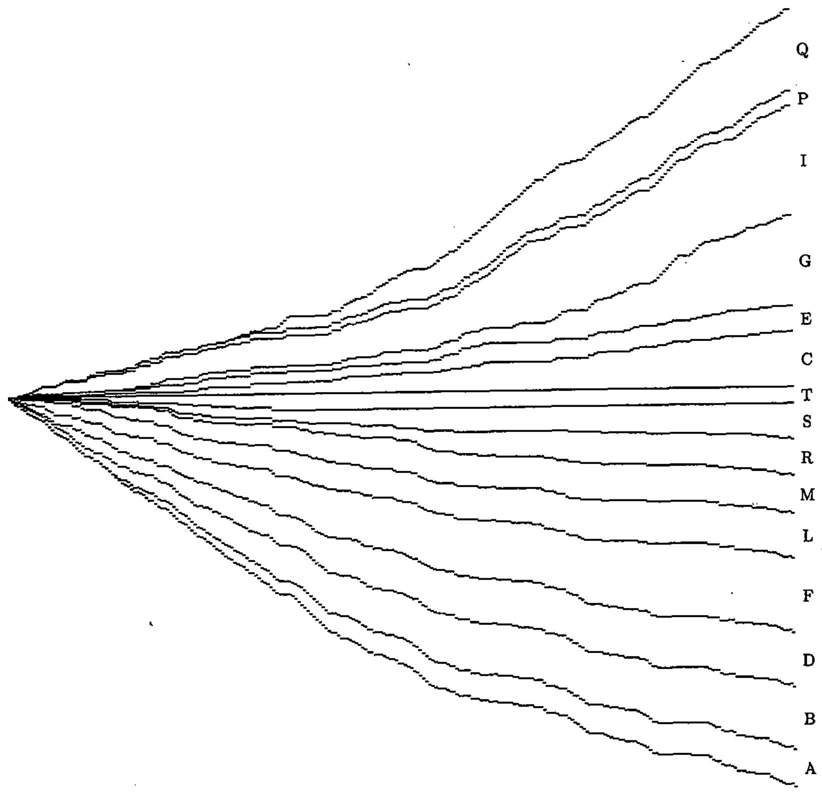

Figure 5. Accumulated errors of TEX78, divided into 15 categories.

with great difficulty. The program now warns any readers who seek to improve TEX to "think thrice before daring to make any changes here." Some indications of my struggles with this particular logic appear in errors #75, #93, and #506.

TEX's line-breaking algorithm determines the optimum sequence of breaks for each paragraph, in the sense that the total "demerits" are minimized over all feasible sequences of breaks. The original algorithm was fairly simple, but it continued to evolve as I fiddled with the formula used to calculate demerits. Demerits are based on the "badness" b of the line (which measures how loose or tight the spacing is) and the "penalty" p for the break (which may be at a hyphen or within a math formula). A penalty might be negative to indicate a good break. The original formula for demerits in TEX78 was

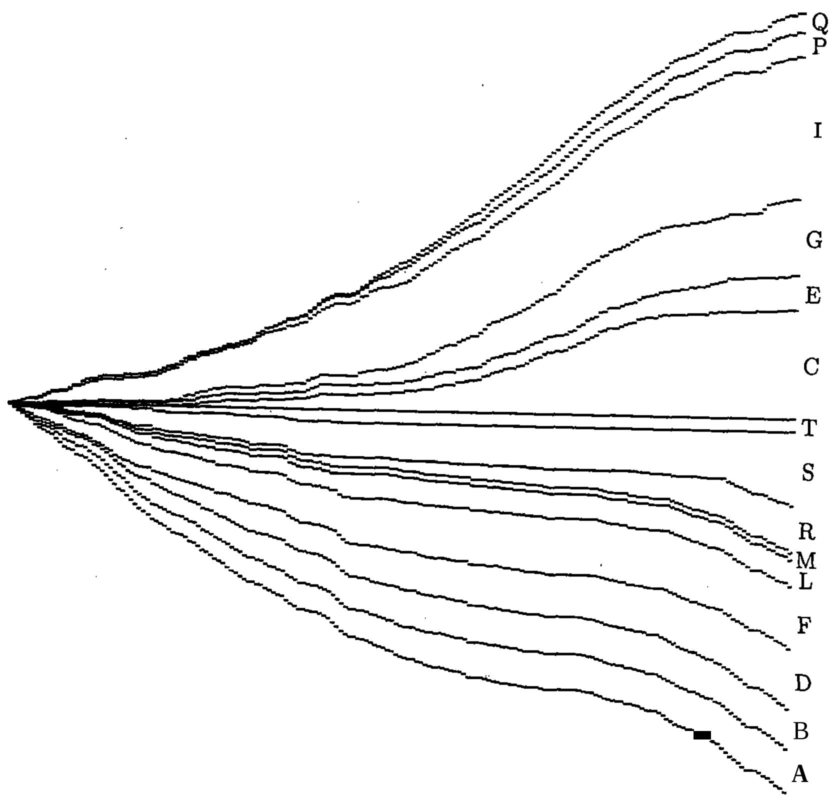

Figure 6. Accumulated errors of TEX82.

$$D = \max(b + p, 0)^2\,;$$

error #76 replaced this by

$$D = \begin{cases} (1 + b + p)^2, & \text{if } p \ge 0; \\ (1 + b)^2 - p^2, & \text{if } p < 0. \end{cases}$$

The extra constant 1 was used to encourage paragraphs with fewer lines; the subtraction of p^2 when $p < 0$ gave fewer demerits to good breaks. This improved formula was published on page 1128 of the article on line-breaking by Knuth and Plass [16]. The first draft of TEX82 added an obvious generalization to the improved formula by introducing a

\linepenalty parameter, l, to replace the constant 1. A further improvement was made in change #554, when I realized that better results would be obtained by computing demerits as follows:

$$D = \begin{cases} (l+b)^2 + p^2, & \text{if } p \geq 0; \\ (l+b)^2 - p^2, & \text{if } p < 0. \end{cases}$$

Otherwise, a line with, say, $(b,p) = (50,100)$, followed by a line with $(b,p) = (0,0)$, would be considered inferior to a pair of lines with $(b,p) = (0,100)$ and $(100,0)$, although the second pair of lines would actually look much worse.

B, Blunders.

A typical blunder, among the 50 or so errors of class B in the appendix, is illustrated by errors #7 and #92. I had declared two symbolic constants in my program, new-line (for one of the three states of TEX's lexical scanner) and next-line (for the sequence of ASCII codes carriage-return and line-feed, needed in SAIL output conventions). Although the meanings were quite dissimilar, the names were quite similar; therefore I confused them in my mind. The compiler didn't detect any syntax error, because both were legal in an output statement, so 1 had to detect and correct the bugs myself. I could have avoided these errors by using a name like *cr_lf* instead of next-line; but that sounds too jargony. A better alternative would have been *new_line_state* instead of new-line.

D, Data disasters.

My most striking error in data-structure updating was #630, which crept in when I made change #625. The error needs a bit of background information before I can explain it: Using an idea of Luis Trabb Pardo, 1 was able to save one bit in each node of TEX's main data structures by putting the nodes in which the bit would be 0—the so-called charnodes—into the upper part of the mem array, all other nodes into the lower part. (It was very important to save this bit, because I needed at least 32 additional bits in every charnode.) One of the aspects of change #625 was to optimize my data structure for representing mathematical subformulas that consist of a single letter. I could recognize and simplify such a subformula by looking for a list that consisted of precisely two elements, namely a charnode followed by a "kern node" (for an *italic correction"). A kern node is identified by (a) not being a charnode, i.e., not having a high memory address, and (b) having the subfield type = 11.

I forgot to test condition (a). But my program still worked in almost every case, because unsuitable lists of length 2 are rare as subformulas, and because the *type* subfield of a charnode records a font number. Amazingly, however, within one week of my installing change #625, some user happened to create a math list of length 2 in which the second element was a character from font number 11!

This example demonstrates that I was lucky to have a wide variety of users. Still, such a bug might survive for years before it would cause trouble for anybody.

F, Forgetfulness.

As I'm writing this paper, I'm trying to remember all the points I wanted to explain about TEX's evolution. Probably I'll forget something, as I did when I was writing the program for TEX.

Usually a bug of class F was easily noticed when I first looked at the corresponding part of the code, with my walk-through-in-execution-order method of debugging. But I'd like to mention two of the F errors that were among the most difficult to find. Both of them occurred in routines that had worked correctly the first few times they were exercised; indeed, these routines had been called hundreds of times, with perfect results, so I no longer suspected that they could be the source of any trouble.

Error #91 occurred in the memory allocation subroutine, the first time I ran out of memory. That subroutine had the general form

```
begin (Get ready to search);
repeat (Look at an available slot);
    if (big enough) then goto found;
    (Move to next slot);
until (back at the beginning);
found: (Allocate and return, unless the available list
    becomes exhausted);
ovfl: (Give an overflow message);
end
```

The bug is obvious: I forgot to say '**goto** *ovfl*' just before the label 'found:'. And it's also obvious why this bug was hard to find: I had lost my suspicions that this subroutine could fail, but when it did fail it allocated one node right in the middle of another. My linked data structure was therefore destroyed, but its defective fields did not cause trouble until several hundred additional operations had been performed by the parts of the program where I was still looking for bugs.

'Error **#203** was even more difficult to find; it lurked in TeX's get-next routine, the subroutine that is executed far more than any other. Whenever TeX is ready to see another token of input, get-next comes into action. Therefore, by the time I had corrected 200 errors, get-next had probably gotten the correct next token more than 100,000 times; I considered it rock-solid reliable.

Since get-next is part of TeX's "inner loop," I had wanted it to be efficient. Indeed, I learned later that the very first statement of get-next, 'cur-cs ← 0', is performed more often than any other single statement of TeX82. (Empirical tests covering a period of more than a year show that 'cur-cs ← 0' was performed more than 1.4 billion times on Stanford's SUAI computer. The get-avail routine, which is next in importance, was invoked only about 438 million times.) Knowing that get-next was critical, I had tried to avoid performing 'cur-cs ← 0' in my first implementations, in cases where I knew that the value of cur-cs would not be examined by the consumers of get-next's tokens. In fact, I knew that cur-cs would be irrelevant in the vast majority of cases. (But I also knew, and forgot, Hoare's dictum that premature optimization is the root of all evil in programming.)

Well, you can almost guess the rest. When I corrected my serious misunderstanding of alignments, errors **#108** and **#110**, I introduced a new case in *get_next*, and that new case filled my thoughts so much that I forgot to worry about the 'cur-cs ← 0' operation. Still, no harm was done unless cur-cs was actually being looked at; TeX wouldn't fail unless \cr occurred in an alignment having a special sort of template that required backup in the parser. As before, the effect of this error was buried in a data structure, where it remained hidden until much later. I found the bug only by temporarily inserting new code that continually monitored the integrity of the data structures. (Such code later became a standard diagnostic feature of TeX82; it can be seen for example in §167.)

L, Language lossage.

Some of my errors (**#98**, **#295**, **#296**, **#480**) were due to the fact that algorithms involving floating-point numbers sometimes fail because of roundoff errors. (I have assigned these errors to class L instead of class A, although it was a close call.) TeX82 was designed to be portable so that it gives essentially identical results on all computers; therefore I avoided floating-point calculations in critical parts of the new program.

Two other errors in my log belong unambiguously to class L: In **#63** and **#827**, I failed to insert parentheses into a macro definition. As a result, when I used the macro with text replacement, any frequent user of macros can guess what happened. (Namely, in **#827**, I had declared the macro

$$\mathit{hi\_mem\_stat\_min} \equiv \text{mem-top} - 13$$

and used it in the statement

$$\mathit{dyn\_used} \leftarrow \text{mem-top} + 1 - \text{hi-mem-stat-min};$$

this gave a minus where I wanted a plus.)

M, Mismatches.

When I write a program I tend to forget the exact specifications of its subroutines. One of my frequent flubs is to blur the distinction between an object and a pointer to that object. In TeX78, for example, I noticed when I got to error **#79** that I had called *vpackage*$(p,\ldots)$ where p pointed to the first node of a vlist, while in the declaration of upackage I had assumed parameters of the form $(h,\ldots)$ where h points to a list header; thus, *link*(h), not h itself, was assumed to point to the first list item. The compiler didn't catch the error because both h and *link*(h) were of type pointer.

While fixing this bug it occurred to me that vpackage was an oft-used subroutine and that I might have made the same mistake more than once. So I looked closely at each of the 26 places I had called vpackage, and the results proved that I was remarkably inconsistent: I had specified a list head 14 times, and a direct pointer 12 times! (Fortunately there wasn't a 13–13 split; that would have been unlucky.)

This error reminded me that I should always check the entire program whenever I notice a mistake; failures tend to recur. In fact, several errors of TeX82 (**#803**, **#813**, **#815**, **#837**) were first noticed when I was debugging similar portions of METAFONT.

R, Robustness.

Most of the changes of type R were introduced to keep TeX from crashing when users supply input that doesn't obey the rules. But some of the R's in the log are intended to keep TeX alive even when other parts of TeX are failing, because of my programming errors or because somebody else is trying to produce a new modification of TeX.

Thus, for example, in #99 and **#123**, I redesigned two of my procedures so that they would produce a symbolic printout of given data structures in memory even when those data structures were malformed. I made it possible to get meaningful output from arbitrary bit configurations in memory, so that while debugging TEX I could look interactively at garbage and guess how it might have arisen.

One of the most recent changes to TEX, #846, has the same flavor: The parameter to show-node-list was redeclared to be of type integer instead of type pointer, because buggy calls on show-node-list might not supply a valid pointer.

S, Surprises.

The most serious errors were those due to my global misunderstandings of how the system fits together. The final error in TEX78 was of type S, and I suppose the final error of TEX82 will be yet another surprise.

Let me mention just two of these. The first is extremely embarrassing, but it makes a good story. TEX produces DVI files as output, where DVI stands for DeVice Independent. The DVI language is like a machine language, consisting of 8-bit instruction codes followed in certain cases by arguments to the instructions. Two of the simplest instructions of DVI language are *push* (code 141) and pop (code 142). It turns out that TEX might output push followed immediately by pop in various circumstances, and this needlessly clutters up the DVI file; so I decided to optimize things a bit by checking to see whether the final byte in my output buffer was push before TEX would output a pop. If so, I could cancel both instructions. This technique even made it possible to detect and cancel long redundant sequences like push push pop push push pop pop pop. Naturally, I checked to see that the buffer hadn't been entirely cancelled out when I tested for such an optimization. (I wasn't 100% stupid.) But I failed to realize that the byte just preceding pop might just happen to be 141 (the binary code for push) when it was the final operand byte of some other instruction. Ouch!

The other S bug I want to discuss is truly an example of global misunderstanding, because it arose in connection with my misperceptions about \global definitions in TEX documents. Users can define control sequences like \abc inside a TEX "group," which is essentially a "block" in the sense of Algol scope rules. At the end of a group, local definitions are rescinded and control sequences revert to the meanings they had at the beginning of the group. In my first implementation of TEX78 I went even further: If \abc was defined inside a group but not before the group had begun, I actually removed \abc from the hash table when the group ended.

There is one exception, however, to TEX's local scope rules (and it's usually the exceptions that lead to surprises). Users can state that a definition is \global; this means that the new definition will survive at

the end of the current group, unless it has been globally redefined again. Therefore my implementation removed control sequences from the hash table at group endings only when they had not been globally defined.

That caused bug #422, which was identical to one of the first serious bugs I had ever encountered when learning to program in the 50s: Deletions from an "open" hash table might make other keys inaccessible, unless the deletions occur in FIFO order, or unless the deletion algorithm takes special precautions to relocate keys in the table. (See my book Sorting and Searching [11], pages 526–527, where I say—in italics—"*The* obvious way to delete records from a scatter table doesn't work.") Alas, I had deleted the control-sequence records in the "obvious way" in TEX78, not realizing that global definitions destroyed the FIFO order.

To fix bug #422, I couldn't patch the definition procedure by using Algorithm 6.4R from my book [11], because the organization of TEX did not allow for relocation of keys. So I needed to change the hash table algorithm from linear probing to chaining, which supports arbitrary deletions. This change was not as painful as it might have been at this late date (August 1979), because I had needed an excuse anyway to overcome my initial hash table design. In order to keep the original implementation simple, I had decided to require that control sequence names be essentially unique when restricted to their first six letters. Such a restriction was quite reasonable when I was to be the only user of TEX; but it was becoming intolerable when the number of users began to grow into the thousands. Therefore change #422 not only altered the hash discipline, it also changed the entire representation mechanism so that identifiers of arbitrary length could be accommodated.

And that wasn't the end of the story. Another year and a half went by before I realized (in #493) that TEX allows declarations like

```
\def\abc{...}
\global\def\xyz{...\abc...}
```

within a group. In such cases I could not eliminate \abc from the hash table at the end of the group, because a reference to \abc still survived within \xyz. I finally decided not to delete anything from the hash table (although I did provide a mechanism to prevent unwanted keys from ever getting in; see #294 and #769).

How did such serious bugs remain undetected for so long? They lay dormant because normal usage of TEX does not require complicated interactions between local and global definitions in groups. Most formatting is simpler than this; even complex books such as The Art of Computer Programming and the TEX manual itself do not need such generality. But if I had used the TRIP test methodology in the early days, 1 would have found and corrected the local/global problems right at the start. This experience suggests that all software systems be subjected to the meanest, nastiest torture tests imaginable; otherwise they will almost certainly continue to exhibit bugs for years after they have begun to produce satisfactory results in large applications.

T, Typographic trivia.

The typographic errors of TEX weren't especially significant, but I'll mention two of them (#69 and #86), where my original SAIL code looked like this:

```
glueshrink(q)←glueshrink(q)←glueshrink(t);
x←x←width(q).
```

SAIL was written for the extended ASCII character set that once was widely used at Stanford, MIT, CMU and a few other places; one of the important characters was '←', for Algol's ':='. The language allowed multiple assignment, hence both of these statements were syntactically correct (although rather silly).

A language designer straddles a narrow line between restrictiveness and permissiveness. If almost every sequence of characters is syntactically correct, the inevitable typographic errors will almost never be detected. But if almost no sequences of characters are syntactically correct, typing becomes a real pain.

In TEX78 I made a terrible decision (#402) to allow users to type a letter like 'A' whenever TEX was expecting to see a number; the meaning was to use the ASCII code of A (97) as the number. This extended the language for certain hacker-type applications; but it caused all sorts of grief to ordinary users, because their typographical errors were being treated as perfectly meaningful TEX input, and they couldn't figure out what was going wrong. (1compounded the error in #507; see also #511. This is a sorry part of the record.) TEX82 resolved the problem by using a special character to introduce ASCII constants.

Some Noteworthy Enhancements

Let's turn now to the other six kinds of errors in the log.

C, Cleanups.

The stickiest issue in TEX has always been the treatment of blank spaces. Users tend to insert spaces in their computer files so that the files look nice, but document processors must also treat spaces as objects that

appear in the final output. Therefore, when you see documents nowadays that have been prepared by systems other than TEX, you often find cases where double spaces appear incorrectly between words; and when you see documents prepared with TEX, you run into cases where a necessary space between words has disappeared. I kept searching for rules that would be simple enough to be easily learned, yet natural enough that they could be applied almost unconsciously. I finally concluded that no such rules existed, and I opted for the best compromise I could find.

Several of the log entries refer to the question of optional spaces after a macro definition. In #133, I decided to ignore a space that appears there; this was prompted by experiences recorded in my comments following #115 and #119. But #133 caused a timing problem in #560, because the macro definition hadn't been fully processed when TEX wanted to check for the optional space; if the user invoked the macro immediately, instead of putting a space there, TEX wasn't ready to respond. Finally in #606 I came to the conclusion that TEX users will best be able to keep their sanity if I do not ignore spaces after definitions; then dozens of similar-appearing cases all have consistent rules.

(See also #220, for space after '$$'; #361, #708, #720, and #723, for space after constants; #440, for space after active characters; and #632, for space after '\\'.)

G, Generalizations.

TEX continued to grow new capabilities as people would present me with new applications. When I couldn't handle the new problem nicely with the existing TEX, I usually would end up changing the system. (But I kept the changes minimal, because I always wanted to finish and get on with other things. More about that later.)

Such generalizations were often built incrementally on the shoulders of their predecessors. For example, the original TEX78 had \output and \mark and macro definitions, which scanned and remembered lists of tokens, but there was no good way to assign a list of tokens to a "token list variable" without causing macro expansion. Then TEX82 added a feature called `\everypar`, which Arthur Keller had long been lobbying for. One day I noticed that I could solve a user's problem in a tricky way by temporarily using `\everypar` to store a list of tokens. This was quite different from the intended use of `\everypar`, of course; so I introduced a new primitive operation called \tokens for such purposes (#559). Later, `\everypar` spawned several descendants called `\everymath` and `\everydisplay` (#568), `\everyhbox` and `\everyvbox` (#649), \everyjob (#657), `\everycr` (#688). I eventually found applications where \tokens wasn't enough by itself and I needed to borrow one of the \every features temporarily to do some nonstandard hackery. So I finally replaced \tokens by an array of 256 registers called **\toks** (#713), analogous to TEX's existing arrays of registers for integers, dimensions, boxes, and glue. TEX82 also acquired the ability to make assignments between different kinds of token-list variables (#746). In such ways I tried to keep the design "orthogonal" as the language grew.

Of course every language designer likes to keep a language simple by applying Occam's razor. I was pleased to discover early in 1977 that simple primitive operations involving boxes, glue, and penalties could account for many of the fundamental operations of typesetting. This was a real unification of basic principles, and it turned out to be even better when I realized that the concepts of ordinary line-breaking applied also to tasks that seemed much harder [16]. But I also fooled myself into thinking that TEX had fewer primitives than it really did, by "overloading" operations that were essentially independent and calling them single features.

For example, my original design of TEX78 would break paragraphs into lines by ignoring all lines whose badness exceeded 200. Later (#104) I made this threshold value user-settable by introducing a new primitive called **\jpar**. Setting **\jpar=2** was something like setting **\tolerance=200** in TEX82; but I also included a peculiar new convention: If **\jpar** was odd, the paragraphs would be set with ragged right margins, otherwise they would be justified to the full width!

Thus, in my attempt to minimize primitives, I had loaded two independent ideas onto a single parameter. I also had packed a half-dozen different kinds of diagnostic output into a single number called \tracing (see #199), whose binary digits were examined individually when TEX was deciding whether to trace parts of its operations.

Then I began to see the need for more user-settable numbers, and I shuddered to think at the resultant multiplicity of new primitives. So I replaced both **\jpar** and \tracing by a single primitive called **\chpar** (#244); one could now say, for example, **\chpar1=2** instead of **\jpar=2**. This change gave me the courage to add new parameters for hyphen penalties, etc., and 1 even added a new parameter to control the raggedness of right margins (#334). Now the parity of **\jpar** was irrelevant; henceforth, the right margins could be either straight or ragged, or they could be produced using some smoothly varying compromise between those extremes—'one third of the way to full raggedness."

My decision to introduce **\chpar** in TEX78 wasn't too bad, because TEX is a macro language and I immediately could define **\jpar** and

\tracing as abbreviations for **\chpar1** and **\chpar2**. But still, those arbitrary numeric codes were inelegant. TEX82 now has fifty different primitive operations that denote integer-valued parameters, each with standard (but user-changeable) names. The old **\jpar** has become \tolerance and **\pretolerance**. The old \tracing has been unbundled into **\tracingparagraphs**, **\tracingpages**, **\tracingmacros**, and a half-dozen more, with separate parameters like **\showboxdepth** to govern the amount of display.

I, Interactions.

About 15% of the errors in the TEX log have been classified type I. The main issue in such cases is to help users identify and recover from errors in their source programs, and this is always problematical because there are so many ways to make mistakes. "When your error is due to misunderstanding rather than mistyping, ... TEX can only explain what looks wrong from its own viewpoint; such an explanation is bound to be mysterious unless you understand the machine's attitude." [21] Which you don't.

Still, I kept trying to make TEX respond more productively, and every such change was logged as an "error" in my original design. The most memorable error of this type was probably #213, when I first realized how nice it would be if I could insert a token or two that TEX could read immediately, instead of aborting a run and starting from scratch. (This was soon followed by #242, when deletion of tokens was also allowed in response to an error message.) I would never have thought of these improvements if I hadn't participated in the implementation and testing of TEX, and I have often wished for similar features in other software I've used since. This one feature must have saved me hundreds of hours as a TEX user during recent years.

Another improvement in interaction didn't occur to me until several months and several hundred pages of output later. Error #338 records the blessed day when I gave TEX the ability to track "runaways," parts of the program that were being processed in the wrong mode because of missing right delimiters. (Further refinements to that change were logged as entries #344, #426, and #793.) Without such provisions, errors that TEX could not have detected until long after their appearance would have been much harder to track down.

There was another significant improvement in interaction that never made it into my error log, because I included it in the original TEX82 without ever putting it into TEX78. This is the *short_display* procedure, for showing the contents of "overfull boxes" and such things in an abbreviated form easily understood by novice users. The *short-display* idea was invented by Ralph Stromquist, who installed it in his early version of TEX at the University of Wisconsin.

P, Portability.

The first changes of type P were simply enhancements to the comments in my SAIL program, but the advent of WEB made it possible for TEX to become truly independent of the machine and operating system it was being run on.

Change **#633** is perhaps the most instructive class-P modification: I decided to guarantee compatibility between DEC-like systems (which break the source file into lines according to the appearance of ASCII *carriage-return* characters) and IBM-like systems (which have fixed-length source lines reminiscent of 80-column cards),[2] in the following way: Whenever TEX reads a line of input, on any system, it automatically removes all blank spaces that appear at the right end. The presence or absence of such blanks therefore cannot influence the behavior of TEX in any way: An ASCII file whose lines are at most 80 characters long (as defined by carriage returns, with or without blanks in front of those carriage returns) can be converted to a file of 80-character records that will produce identical results with TEX, simply by padding each line with blanks.

Change **#791** carried **#633** to its logical conclusion.

Q, Quality.

From the beginning, I wanted TEX to produce documents of the highest possible typographic quality. The time had come when computer-produced output no longer needed to settle for being only "pretty good"; I wanted to equal or exceed the quality of the best books ever printed by photographic methods.

As Kernighan and Cherry have said, "The main difficulty is in finding the right numbers to use for esthetically pleasing positioning. ...Much of this time has gone into two things—fine-tuning (what is the most esthetically pleasing space to use between the numerator and denominator of a fraction?), and changing things found deficient by our users (shouldn't a tilde be a delimiter?)." [8]

[2] Paradoxically, DEC has also introduced the VMS operating system, which has fixed-length lines that can include troublesome carriage-returns. But that's another story.

I too had trouble with numerators and denominators: Change **#229** increased the amount of space surrounding the bar line in displayed fractions, and I should have made a similar change to fractions in text. (Page 68 of the new Volume 2 turned out to be extremely ugly because of badly spaced fractions.) TEX82 was able to improve the situation because of my experiences with TEX78, but even today I must take special precautions in my TEX documents to get certain fractions and square roots to look right.

The Evolution Process as a Whole

Looking now at the entire log of errors, I'm struck by the fact that my attitude during those years was clearly far from ideal: My overriding goal was always to finish, to finish, to get this long-overdue project done so that I could resume work on other long-overdue projects. I never wanted to spend extra time studying alternatives for the best possible typesetting language; only rarely was I in a mood to consider any changes to TEX whatsoever. I wanted TEX to produce the highest quality, sure, but I wanted to achieve that with the minimum amount of work on my part.

At the end of almost every day between March 29, 1978, and March 29, 1980, I felt that TEX78 was a complete system, containing no bugs and needing no further enhancements. At the end of almost every day since September 9, 1982, I have felt that TEX82 was a complete system, containing no bugs and needing no further enhancements. Each of the subsequent steps in the evolution of TEX has been viewed not as an evolutionary step towards a vague distant goal, but rather as the final evolutionary step towards the finally reached goal! Yet, over time, TEX has changed dramatically as a result of many such "final steps."

Was this horizon-limiting attitude harmful, or was it somehow a blessing in disguise? I'm pleased to see that TEX actually kept getting simpler as it kept growing, because the new features blended with the old ones. I was constantly bombarded by ideas for extensions, and I was constantly turning a deaf ear to everything that didn't fit well with TEX as I conceived it at the time. Thus TEX converged, rather than diverged, to its final form. By acting as an extremely conservative filter, and by believing that the system was always complete, I was perhaps able to save TEX from the "creeping featurism" [24] that destroys systems whose users are allowed to introduce a patchwork of loosely connected ideas.

If I had time to spend another ten years developing a system with the same aims as TEX—if I were to start all over again from scratch, without any considerations of compatibility with existing systems—I could no doubt come up with something that is marginally better. But at the moment I can't think of any big improvements. The best such system I can envision today would still look very much like TEX82; so I think this particular case study in program evolution has proved to be successful.

Of course I don't mean to imply that all problems of computational typography have been solved. Far from it! There still are countless important issues to be studied, relating especially to the many classes of documents that go far beyond what I ever intended TEX to handle.

Conclusions

My purpose in this paper has been to describe what I think are the most significant aspects of the experiences I had while developing TEX, basing this on a study of more than 800 errors that I noted down in log books over the years. I've tried to interpret many specific facts and observations in a sufficiently general way that readers may understand how to apply similar concepts to other software developments.

In Volume 1 of The Art of Computer Programming [9], I wrote:

> Debugging is an art that needs much further study ...The most effective debugging techniques seem to be those which are designed and built into the program itself ...Another good debugging practice is to keep a record of every mistake that is made. Even though this will probably be quite embarrassing, such information is invaluable to anyone doing research on the debugging problem, and it will also help you learn how to reduce the number of future errors.

Well, I hope that my error log in the appendix below, especially the first 237 items (which relate specifically to debugging), will be useful somehow to people who study the debugging process.

But if you ask whether keeping such a log has helped me learn how to reduce the number of future errors, my answer has to be No. I kept a similar log for errors in METAFONT, and there was no perceivable reduction. I continue to make the same kinds of mistakes.

What have I really learned, then? I think I've learned, primarily, to have a better sense of balance and proportion. I now understand the complexities of a medium-size software system, and the ways in which it can be expected to evolve. I now understand that there are so many kinds of errors, we cannot stamp them out by systematically eliminating everything that might be "considered harmful." I now understand enough about my propensity to err that I can accept it as an act of life; I now can be convinced more easily of my fallacy when I have made a mistake. Indeed, I now strive energetically to find faults in my own work,

even though it would be much easier to look for assurances that everything is OK. I now look forward to making (and correcting) hundreds of future errors as I write Volume 4 of The Art of Computer Programming.

Addendum: Fifteen Months More

As mentioned above, I began to write this paper in May of 1987, but I decided to wait before publication until more time had gone by. Then I could present a "complete" and "final" record of TEX's errors.

Now it's September, 1988, and I've decided to bring this paper to a possibly premature conclusion, because I'm scheduled to present it at a conference [4]. TEX still hasn't shown encouraging signs of becoming quiescent; indeed, sixteen more entries have entered the error log since May, 1987, including three as recent as June, 1988. Therefore it still isn't the right moment to manufacture TEX on a chip!

All errors known to me as of September 1, 1988, are now included in the appendix to this paper; the total has now reached 865.[3]

I plan to publish a brief note ten years from now, bringing the list to its absolutely final form.

I have been paying a reward to everyone who discovers new bugs in TEX, and doubling the amount every year. Last December I made two payments of $40.96 each, and my checkbook has been hit for five $81.92 payments in recent months. I'm desperately hoping that this incentive to discover the final bugs will produce them before I am unable to pay the promised amount. (Surely in 1998 I won't be writing checks for $83,886.08?)

As I expected, half of the most recent errors have fallen into the surprise (S) category—even though surprises, by definition, are unexpected. But one of the others (error #854) was perhaps the most surprising of all, because it was the result of a terrible algorithm by a person who certainly should have known better (me). I wanted to multiply the two's-complement fixed-point number

$$A = -16 + a_1 \cdot 2^{-4} + a_2 \cdot 2^{-12} + a_3 \cdot 2^{-20}, \qquad 0 \le a_i < 256,$$

by the positive quantity $z/2^{16}$, where z is an integer, $2^{26} \le z < 2^{27}$, obtaining an answer of the form $P/2^{16}$ where P is an integer, $|P| < 2^{31}$;

[3] **(Footnote added September** 1991) **In fact we are now up to** 916, **primarily because of major changes in** 1989 **that can be said to have inaugurated "Phase 4" of** TEX82.

all intermediate quantities in the calculation were required to be less than 2^{31} in absolute value. My program did this by computing

```
$C \leftarrow 16 * Z$;
$Z \leftarrow Z$ div 16;
$P \leftarrow ((a_3 * Z)$ div $256 + a_2 * Z)$ div $256 + a_1 * Z - C$;
```

I should rather have computed

```
$Z \leftarrow Z$ div 16;
$P \leftarrow ((a_3 * Z)$ div $256 + a_2 * Z)$ div $256 + (a_1 - 256) * Z$.
```

(Consider, for example, the case $Z = 2^{26} + 15$ and $a_1 = a_2 = a_3 = 255$, so that $A = -2^{-20}$. The first method gives $P = -304$; the second method gives the correct answer, $P = -64$.)

Let me close by discussing one more recent error, #864. This change yields only a slight gain in efficiency, so I needn't have made it; but it was easy to correct one more statement while I was fixing #863. It's an instructive example of how a design methodology based on invariants might not lead to the best algorithm unless we think a bit harder about what is going on.

Here's the idea: Each run of TEX determines a threshold value 8 above which the (one-word) charnodes will reside, below which all other (variable-size) nodes will be stored. Actually there are two values, θ_0 and θ_1; memory positions between θ_0 and θ_1 are unused. (In the program, θ_0 is actually called *lo_mem_max*, and θ_1 is called *hi_mem_min*.) TEX changes θ_0 and θ_1 conservatively as it runs, so that they will converge to values appropriate to particular applications. The boundary value θ was originally fixed at compile time; this transition to "late binding" was change #819.

When TEX needs more space for charnodes, it usually sets $\theta_1 \leftarrow \theta_1 - 1$; when TEX needs more space for variable-size nodes, it usually sets $\theta_0 \leftarrow \theta_0 + 1000$. But we need to have $\theta_0 < \theta_1$. Therefore, instead of setting $\theta_0 \leftarrow \theta_0 + 1000$, my original code said

```
if $\theta_1 - \theta_0 > 1000$ then $\theta_0 \leftarrow \theta_0 + 1000$
else if $\theta_1 - \theta_0 > 2$ then $\theta_0 \leftarrow (\theta_0 + \theta_1 + 2)$ div 2
else (Report memory overflow).
```

(The variable θ_0 had to increase by at least 2.) Chris Thompson of Cambridge University pointed out that this strategy, while preserving the necessary invariants, is discontinuous. If $\theta_1 - \theta_0 = 1001$, the algorithm gobbles up all the discretionary space that's left. Therefore change #864 substituted better logic:

```
if $\theta_1 - \theta_0 \geq 1998$ then $\theta_0 \leftarrow \theta_0 + 1000$
else if $\theta_1 - \theta_2 > 2$ then $\theta_0 \leftarrow \theta_0 + 1 + (\theta_1 - \theta_0)$ div 2
else (Report memory overflow).
```

The new version also avoids problems on certain computers when θ_0 and θ_1 are negative; that was error #863. (Of course, when TEX is this close to running out of memory, it probably won't survive much longer anyway. I'm grasping at straws. But I might as well grasp intelligently.)

Acknowledgments

I've already mentioned that the TEX project has had hundreds of volunteers who helped to guide me through all these developments. Their names can be found in the rosters of the TEX Users Group; I couldn't possibly list them all here. Luis Trabb Pardo and David R. Fuchs were my "right-hand men" for TEX78 and TEX82, respectively. The project received generous financial backing from several independent sources, notably the System Development Foundation, the National Science Foundation, and the Office of Naval Research. The material on which this report has been based is now housed in the Stanford University Archives; I wish to thank the archivist, Roxanne L. Nilan, for her friendly cooperation. Thanks are due to the referee who helped me to remove errors not from TEX but from this paper. And above all, I want to thank my wife, Jill, for ten years of exceptional tolerance; software development is much more demanding than the other things I usually do. Jill also helped me to design the format for the appendix that follows.

References

[1] Victor R. Basili and Barry T. Perricone, "Software errors and complexity: An empirical investigation," Communications of the *ACM* 27 (1984), 42–52.

[2] Beeton, Barbara [Ed.], *TEX* and METRFONT: Errata and Changes, 09 September 1983, distributed with *TUGboat* 4 (1983).

[3] L. A. Belady and M. M. Lehman, "A model of large program development," IBM Systems Journal 15 (1976), 225–252.

[4] Reinhard Budde, Christiane Floyd, Reinhard Keil-Slawik, and Heinz Ziillighoven [Eds.], Software Development and Reality Construction (Berlin: Springer, 1991), in press.

[5] A. Endres, "An analysis of errors and their causes in system programs," Proceedings of an International Conference on Software Engineering (1975), 327–336.

[6] David R. Fuchs and Donald E. Knuth, "Optimal prepaging and font caching," *ACM Transactions* on Programming Languages and Systems 7 (1985), 62–79.

[7] Piet Hein, Grooks (Cambridge, Massachusetts: MIT Press, 1966).

[8] Brian W. Kernighan and Lorinda L. Cherry, "A system for typesetting mathematics," Communications of the *ACM* **18** (1975), 151–157.

[9] Donald E. Knuth, Fundamental algorithms: The Art of Computer Programming **1** (Reading, Massachusetts: Addison-Wesley, 1968), xxii + 634 pp. Second edition, 1973.

[10] Donald E. Knuth, Seminumerical Algorithms: The Art of Computer Programming 2 (Reading, Massachusetts: Addison-Wesley, 1969), xii + 624 pp. Second edition, 1981, xiv + 689 pp.

[11] Donald E. Knuth, Sorting and Searching: The Art of Computer Programming 3 (Reading, Massachusetts: Addison-Wesley, 1973), xii + 722 pp.

[12] Donald E. Knuth, "Mathematical typography," Bulletin of the American Mathematical Society (new series) **1** (1979), 337–372.

[13] Donald E. Knuth, *TEX*, a System for Technical Text (Providence, Rhode Island: American Mathematical Society, 1979), 198 pp.

[14] Donald E. Knuth, *TEX* and *METAFONT:* New Directions in Typesetting (Bedford, Massachusetts, Digital Press, 1979), xi + 45 + 201 + 105 pp.

[15] Donald E. Knuth, "The letter S," The Mathematical *Intelligencer* 2 (1980), 114–122.

[16] Donald E. Knuth and Michael F. Plass, "Breaking paragraphs into lines," Software — Practice & Experience **11** (1981), 1119–1184.

[17] Donald E. Knuth, "The concept of a meta-font," Visible Language 16 (1982), 3–27.

[18] Donald E. Knuth, The WEB System of Structured Documentation, Computer Science Department Report STAN-CS-83-980, Stanford University, Stanford, CA (September 1983), 206 pp.

[19] Donald E. Knuth, A torture test for *TEX*, Computer Science Department Report STAN-CS-84-1027, Stanford University, Stanford, CA (November 1984), 142 pp.

[20] Donald E. Knuth, "Literate programming," The Computer Journal **27** (1984), 97–111.

[21] Donald E. Knuth, The *TEXbook* (Reading, Massachusetts: Addison-Wesley, 1984), 483 pp.

[22] Donald E. Knuth, A torture test for *METRFONT,* Computer Science Department Report STAN-CS-86-1095, Stanford University, Stanford, CA (January 1986), 78 pp.

[23] Donald E. Knuth, TEX: The Program: Computers & Typesetting B (Reading, Massachusetts: Addison-Wesley, 1986) xvi + 594 pp.

[24] Guy L. Steele Jr., Donald R. Woods, Raphael **A.** Finkel, Mark R. Crispin, Richard M. Stallman, and Geoffrey S. Goodfellow, Hacker's Dictionary: **A** Guide to the World of Wizards (New York: Harper and Row, 1983).

[25] C. Széchy, Foundation Failures (London: Concrete Publications, 1961).

[26] Patrick Winston, Artificial Intelligence: An MIT Perspective (Cambridge, Massachusetts: MIT Press, 1979).