Ideas for NTS—An Extended TEX Implementation

Nelson H. F. Beebe

Contents

List of Tables

1 Introduction

With the completion of the TEX project, Donald Knuth decreed that TEX and METAFONT are complete [26], and will not be developed further by him, other than fixing increasingly rare bugs. Furthermore, any future systems based on these programs must be called by different names.

Nevertheless, wide experience with TEX up to its most recent incarnation, version 3, has produced numerous requests for enhancements.

At the instigation of Joachim Lammarsch and others, a cooperative effort has begun to consider the design of a system that might someday succeed TEX; the working name chosen is *New Typesetting System* (NTS). This proposed logo recognizes its TEX ancestry, and I suggest that it could be pronounced 'nits', which are laid by gnats on gnus, though some may prefer the softer sound 'ehnts', which is close to 'ants' and preserves the insect traditions of the field.[1]

2 Previous work

One of the first published commentaries on the deficiencies of TEX is the extensive analysis by Mittelbach [30].

The present author responded to Knuth's final words in [12].

Adams has documented deficiencies of the TEX/PostScript/graphics interface [1].

Vulis has suggested enhancements to TEX for support of scalable fonts [42, 43], and provided a working implementation, called Vector TEX, or VTEX, that is well-documented in a book [44].

Asher [8] documents the *Type & Set* system which is used by the publisher Current Science for

[1] Book worms sometimes discover book lice in old texts.

the production of 86 journals. It uses a front-end processor to convert word-processor input into TEX markup, and modifies TEX's back-end DVI output.

Clark questions the advisability of extending TEX [17].

Taylor comments on what might follow TEX [39, 41].

Barr [9] documents shortcomings of TEX for the typesetting of particular areas of mathematics.

Raman [35] considers the problem of typesetting mathematics for sight-impaired users. Perhaps extensions to TEX could be helpful in this effort.

Semenzato and Wang [36] propose a more powerful front-end programming language interface to TEX.

Knuth and MacKay describe TEX--XET [25], a modification of TEX for typesetting of bi-directional text, such as English mixed with Arabic or Hebrew. This requires extensions to the TEX DVI file format, which in turn requires support from DVI drivers. Breitenlohner (????) produced a post-processing filter, `dvidvi`, to convert such extended DVI files to normal ones in order to avoid the need for driver extensions.

Becker and Berry [11] demonstrate an extension to `troff` for tri-directional typesetting (e.g. Chinese, Arabic, and English), and comment on the implications of this for TEX. They report that their extensions to `troff` are incapable of handling quad-directional typesetting, such as might occur if Mongolian, which is read in columns from left to right, were added to a tri-directional document.

Srouji and Berry [38] consider the problem of left-justification of Arabic by elongation of letter forms. Their implementation is based on `troff`, and they describe why TEX is not capable of this.

Plaice [33] describes an extension of TEX called Ω. Two of its important features are the increase in the number of fonts, ligatures, and registers, and support for 16-bit character sets.

Computer language design should always be done with a solid understanding of the issues, which is best gained through a study of the history of programming languages. I urge the NTS design team to review the two History of Programming Language Conference proceedings [23, 45, 24] before starting.

3 My wish list

At the TUG'93 Conference (July 26–30, 1993) at Aston University, Birmingham, UK, Philip Taylor described the current status of the NTS project which he is most capably coordinating and leading [40]. He stated at the outset that because of the widespread use of TEX, both as a means of typesetting text, and as a means of communications of material, particular mathematics, in the limited character sets available on current computers, that it was important for NTS to be a superset of TEX, but acting exactly as TEX, until asked to provide the NTS extensions.

In a July 13, 1992, posting to the NTS-L electronic mail discussion list, Richard Palais argued forcefully for preservation of compatibility with TEX:

> *... every year the American Mathematical Society not only publishes many tens of thousands of pages of books and primary mathematical journals in TEX, it also publishes more tens of thousands of pages of Mathematical Reviews (MR). The cost of producing just one year of MR is well in excess of five million dollars, and all of MR going back to 1959 (about one million records) is stored online* in TEX format *in the MathSci database.*

In making proposals for extensions to TEX, it is therefore important to distinguish between features that can be added, perhaps with considerable difficulty, by TEX macros, and features that will require fundamental changes to TEX itself.

The wish list that follows is my contribution to the collection of ideas about what might be usefully added to a new system that is derived from TEX. The wish list is arranged as a series of independent subsections, in no particular order.

3.1 Implementation limits

The history of computing has repeatedly shown the dangers of designing hardware and software to the limits of current technology.

Patterson and Hennessy [32] document numerous examples; on the subject of memory, they quote Bell and Strecker (p. 481):

> *There is only one mistake that can be made in computer design that is difficult to recover from—not having enough address bits for memory addressing and memory management. The PDP-11 followed the unbroken tradition of nearly every known computer.*

They then go on to observe:

> *A partial list of successful machines that eventually starved to death for lack of address bits includes the PDP-8, PDP-10, PDP-11, Intel 8080, Intel 8086, Intel 80186, AMI 6502, Zilog Z80, Cray 1, and Cray X-MP.*

The year 1992 saw the introduction of 64-bit address spaces in new architectures from DEC and MIPS/SGI. Hewlett-Packard, IBM, and Sun have announced plans to move to 64-bit architectures by

1994 or 1995. At the substantial data rate of 100 MB/sec, it will take about 5000 years to write 2^{64} bytes of memory, so it looks like the address-space limitation will finally be laid to reset.

TEX was created before

- personal computers and workstations,
- bitmapped graphics displays,
- cheap memory and disk,
- fast RISC processors,
- laser printers,
- graphics standards,
- standard floating-point arithmetic,
- PostScript,
- SGML.

Some of these changes are dramatic: since TEX was conceived, CPU performance has increased by two orders of magnitude, and disk and memory costs have fallen by the same factor.

TEX's birth year was a time where high-end computer-based typesetting was usually based on optically-scaled fonts, with only horizontal and vertical orientations of text and rules, no graphics support other than by manual paste-up, and no color without preparation of color separations.

When the initial SAIL-language version of TEX was rewritten to produce the current implementation, Knuth chose Pascal as the target language for portability, and further restricted the Pascal used in TEX and METAFONT to avoid features that would inhibit translation to other languages (e.g. nested procedures), or were unevenly implemented (e.g. the Pascal memory deallocator, `dispose`, was often a dummy routine, so dynamic memory could never be reused), or were poorly supported by the language (notably, character strings).

These limitations (which were quite reasonable at the time) required Knuth to go to great trouble in programming TEX, and in the interests of memory economizations, hard-to-change decisions were made by allocating table sizes in fixed numbers of bits, and memory was handled by private code that used arrays of fixed size set at compile time. Consequently, one of the commonest errors met by large applications is the dreaded `TeX capacity exceeded` message, and inordinate amounts of programmer time have been expended in trying to work around limitations of TEX table sizes, and limits on the number of fonts, font families, and registers.

While some implementations of TEX provide for run-time setting of table sizes (a valid system-dependent extension), none that I know of can dynamically expand tables that overflow, and none can remove the limits imposed by allocation of short bitfields to hold table sizes, such as the limits to 16 font families, 256 registers, and 256-character fonts.

Modern implementation languages provide reliable support for dynamic memory allocation, and with careful encapsulation of table access, it is possible to detect table overflow and provide for dynamic enlargement. This has the serendipitous feature that small jobs then have only small memory requirements, and large jobs never run out of memory until system memory resources are completely exhausted.

I therefore conclude that NTS should remove artificial limitations on table sizes to the greatest extent possible.

3.2 Text shapes

With the `\hang`, `\hangafter`, and `\hangindent` commands, TEX provides for typesetting a paragraph in two blocks with different line widths. With the more general `\parshape` command, it supports typesetting of a paragraph in a single region of arbitrary shape. The problem is that the shape is restricted to the paragraph, and without tricks like separating paragraphs with `{\par}`,[2] the shape is forgotten at the start of the next paragraph.

NTS should generalize this further, allowing the shape to contain multiple paragraphs. Although non-rectangular text shapes are uncommon in most applications, they are frequently required in advertising and magazine typesetting.

The generalization could be through a new command called `\textshape`, with `\parshape` retaining its TEX meaning.

3.3 Improved control of macro expansion

Programming languages with macro languages always raise the issue of the timing of macro expansion. TEX deals with this in a limited way by providing `\edef` to force expansion at macro definition time, and `\noexpand` to suppress expansion. The trouble with the latter is that once `\noexpand` itself is expanded, it is removed from the input stream, so that if the text is subsequently rescanned, the macro that followed `\noexpand` will be expanded. LATEX tries to deal with this by introducing the `\protect`

[2] This curious trick causes the paragraph shape to be discarded inside the group by the actions associated with `\par`, then restored by the normal end-of-group actions.

macro which is redefined according to context to values like `\noexpand \noexpand \noexpand` and `\noexpand \protect \noexpand`; each additional `\noexpand` macro accounts for one more level of expansion.

It seems that it would be useful to have a pair of commands, `\neverexpand` to (almost) permanently protect a following macro from expansion, and `\expandall` to expand all macros in its following braced argument, *including* those protected by `\neverexpand`. Protected macros could then pass through an arbitrary number of scans until the point that expansion was really needed.

3.4 Modules and name spaces

Modern programming languages such as Ada [5], Modula-2 [46], and Fortran 90 [7] allow information hiding to a greater degree than older languages such as Fortran and C by introducing the notion of *packages*, or *modules*. These provide restricted containers for identifiers that hide them from other code. Some of the identifiers can be permanently hidden by declaring them to be private, and the others can be made publicly visible, but only to code that explicitly requests their importation. Some languages even provide for access permissions, so that public variables can be made read-only or write-only outside their use in the package. A name that occurs in two or more packages that are imported by the same piece of code can be disambiguated implicitly by context of use or in the case of functions and procedures, by argument types, or disambiguated explicitly by qualifying the name with a package-name prefix.

Without packages, programmers of languages like Lisp, Pascal, C, and TEX, are forced to adopt variable-naming conventions that explicitly incorporate package names as a prefix or suffix, or special characters that are not normally used by ordinary applications (witness the extensive use of the '`@`' character in TEX macro packages in internal macro names). The trouble with these schemes is that they reduce readability, and they do not really provide protection against accidental reuse of the name.

Like TEX and METAFONT, the UNIX operating system kernel was originally a monolithic program with a single global name space that grew to the point where it was very difficult to maintain and extend. Curiously, at the time of TEX's initial design, all three programs were about the same size. The introduction of network support and a desire for enhanced portability have resulted in more recent versions of UNIX adopting a micro-kernel architecture, with the program being split into several independent pieces that communicate through well-defined interfaces. A similar redesign of IBM's OS/2 is in progress. Packages and modules are important tools for reducing the complexity of large programs.

I would very much like to have a package facility in NTS. One possible syntax would be to have package name declaration and use like

```
\newpackage \PictureMode
\beginPictureMode
...
\def \put (#1,#2)#3{...}
...
\endPictureMode
```

with unqualified use in a group like

```
{
  \with \PictureMode
  ...
  \put(0,0){...}
  ...
}
```

or in an environment like

```
\beginPictureMode
...
\put(10,20){...}
...
\endPictureMode
```

and qualified use like

```
\PictureMode.put(10,20){...}
```

In a package definition, all names would by default be visible if the package were subsequently selected, unless the definition contained explicit restrictions like

```
\private \internalput
\private \internalmultiput
```

to restrict visibility of selected identifiers.

In the presence of declarations like

```
\export \put
\export \multiput
```

no names declared in the package would be visible outside the package, other than those explicitly exported.

Restricted visibility of names in a package could be obtained by syntax like

```
\beginPictureMode
\import \put
\put(10,20){...}
\endPictureMode
```

The presence of `\import` would immediately restrict visibility of names in the package to just those given on `\import` commands.

One could include access permissions for names by declarations like

```
\readonly \x
\writeonly \n
```

and packages could be made unmodifiable by a special declaration as in this example:

```
\beginPictureMode
\restrict
\def \put (#1,#2)#3{...}
\endPictureMode
```

`\restrict` would prevent further modifications to existing macros defined in the package, a feature that would improve robustness of large packages like LaTeX and $\mathcal{AMS}$-TeX. New macros could still be added to the package by users, but those macros would also be restricted.

It appears that a package facility could be introduced without compromising existing documents, because the `\newpackage` macro declares the following name to be of package type, with a corresponding `\beginxxx` and `\endxxx` environment. Subsequent appearance of the package name followed by a dot would unambiguously indicate qualified use of the following name.

3.5 Dictionaries

The PostScript language introduces the interesting concept of dictionaries, and a dictionary stack. Most conventional programming languages have a hierarchy of name spaces, including at least some of these: *block*, *procedure* or *function*, *file*, *package* (or *module*), and *world*. Except for packages, these are strictly nested inside one another, and a procedure cannot refer to a local variable of another procedure.

PostScript dictionaries (called symbol tables in other languages) provide a more general model of name space control. The idea is that the name space can itself have a name, and that name spaces can be explicitly selected by pushing the names onto a dictionary stack which is searched in order from top (last in) to bottom (first in). The anonymous block name scope provided in other languages can be obtained inside `save`/`restore` pairs, very much like TeX braces. Named scopes like functions, files, packages, and world, are provided by named dictionaries. However, entering a PostScript procedure does not automatically select a new name scope like procedure entry does in most other languages; instead, the programmer can explicitly control the name scope by pushing a new dictionary with `begin dictname`, and later popping it with `end`. Unlike `save`/`restore`, dictionary entries are not forgotten when an `end` is popped.

This model is very powerful, and efficient. In most cases, procedures do not require their own dictionaries, so no scope changing code needs to be implemented at procedure entry and exit. When required, procedures can share names and their values by pushing an appropriate dictionary. Name search is always carried out from the top-most dictionary downward, with the lowest entry always being the standard system dictionary. This permits private dictionaries to contain redefinitions of *all* variables, functions, and operators. It is possible to save the current definition of a name before redefining it, just as TeX programmers often write

```
\let \oldfoo = \foo
\def \foo {...\oldfoo...}
```

to augment the action of a macro with additional code.

The initial definition of PostScript (Level 1) required dictionaries to be created with a fixed size; this was later found to be a bad design choice, and in Level 2 PostScript, dictionaries expand automatically.

Dictionaries appear to offer a facility akin to the dynamic scoping of some Lisp dialects (e.g. MacLisp and Emacs Lisp), which Stallman argues [10, p. 311–312] is more powerful than the lexical scoping of most other programming languages, including Common Lisp and Scheme.

I believe that dictionaries could be easily added to NTS, since the stacking name space mechanism is already present in TeX to support unnamed braced groups.

Whether both packages and dictionaries are needed is debatable. PostScript dictionaries provide for explicit name qualifications, albeit via the onerous syntax of `begin dictname opname end` instead of the more compact `dictname.opname` used in packages, but they do not force a name space hierarchy upon the programmer. I'm inclined to believe that both would be valuable additions.

3.6 Floating-point arithmetic

In order to guarantee identical operation across all architectures, Knuth did not make floating-point arithmetic available in the TeX language.

TeX the program has only one variable of floating-point type, and it is used only for glue ratio computations that do not affect line breaking decisions. Some personal computer implementations of TeX replace this by scaled fixed-point arithmetic, since such machines often lack floating-point hardware. METAFONT the program has no floating-point arithmetic whatsoever.

Except for the glue ratios, TEX does all of its calculations in 32-bit integer and scaled integer arithmetic.

Integer counter values range from $-2^{31}+1$ to $2^{31}-1$ (-2147483647 to $+2147483647$).

Dimensions are represented in units of scaled points (65536 sp = 1 pt), so that they are effectively the sum of a 15-bit integer and a 16-bit fraction. However, to allow two dimensions to be added without overflow, the integer part is restricted to 14 bits, so that dimensions actually range from $-2^{30}+1$ sp to $2^{30}-1$ sp (approximately -16384 pt to $+16384$ pt).

At the time TEX was first designed in 1977–1978, there were almost as many floating-point architectures as computer architectures, and with the possible exception of the IBM System/360 mainframes, no one system was dominant. System/360 floating-point arithmetic has two serious deficiencies: hexadecimal normalization leading to wobbling precision, and truncating, rather than rounding, operations. Thus, there was no obvious candidate for designing a software emulation of floating-point arithmetic.

That situation changed in 1980 when a draft specification of a standard binary floating-point arithmetic was published by IEEE Task P754. Although final standardization actually did not happen until 1985 [31], the arithmetic system was implemented in hardware by several vendors, and today is the dominant system used by virtually all personal computers and workstations, with more than 150 million having been sold by 1993.

Experience in the numerical computing community has shown that there are some variations in quality of implementation that prevent achieving bit-for-bit identical results in longer computations; these arise from differences in instruction ordering (recall that because of finite precision and range, floating-point addition and multiplication is not necessarily associative), and from behavior of the floating-point implementations near the underflow and overflow limits.

However, in the context of NTS, these variations would not exist, *provided* that IEEE 754 floating-point arithmetic were implemented entirely in software. This is certainly possible; one excellent example is Apple's SANE (Standard Apple Numeric Environment) implementation used in those models of the Apple Macintosh that lack floating-point hardware. SANE was written by Jerome Coonan, one of the authors of the IEEE 754 Standard.

It does not take much experience with TEX arithmetic to rediscover the lesson learned on the computers of the 1950's that scaled integer arithmetic is exceedingly painful to program in, and even more difficult when double-length products cannot be produced, and overflow cannot be detected.

Although TEX detects integer overflow in multiply and divide instructions, it does *not* do so for addition. I once reported the following behavior in a TEX bug report to Don Knuth:

```
\count0 = 1073741824
\multiply \count0 by 2
! Arithmetic overflow.

\count0 = 1073741824
\advance \count0 by \count0
\showthe \count0
> -2147483648.
```

His response was that it is an implementation 'feature', rather than a bug, and that overflow checks are omitted for addition because they happen in a number of places, and might impact performance.

My personal view is that this behavior is simply *incorrect*: getting the wrong answer fast is always inferior to getting the right answer somewhat more slowly. In any event, machines today are very much faster than those at TEX's birth, and the extra overhead of overflow checking for addition should certainly be tolerable.

I believe that NTS needs both an IEEE 754 floating-point package in software (to guarantee identical results across all architectures, and allow operation on machines that lack floating-point hardware), and a minimal set of elementary functions such as those provided in the Fortran language standards, for which reliable and accurate algorithms have been published by Cody and Waite [18], and implemented in freely- and publicly-available code for Berkeley UNIX.

IEEE 754 support for sticky exception flags, gradual underflow to denormalized numbers, and for NaN (Not-a-Number) and Infinity, make it possible to provide for non-stop computing in the event of underflow and overflow, without sacrificing the ability to detect that abnormal conditions might have occurred during the computation. There is then no need to add the complexity of exception handling to the TEX language.

Applications of floating-point arithmetic become apparent as soon as rotated or scaled text, and graphics, are considered.

The PostScript page description language provides IEEE 754 arithmetic, although only the 32-bit version (the Standard actually defines four formats, but most implementations offer only two or three of

them). Floating-point arithmetic in PostScript implementations in output devices have been in both hardware and software. In the case of hardware, some variations between output devices are therefore possible. Certainly any successor to PostScript will also have IEEE 754 floating-point arithmetic.

3.7 Overflow detection

In Section 3.6 on page 1005, we observed that TEX does not detect integer overflow on addition or subtraction. This should be remedied.

3.8 64-bit arithmetic

The trend in computer architectures of the 1990s is to move beyond the 32-bit address space limit first introduced with the DEC VAX architecture in 1978. Today, the MIPS R4000 and DEC Alpha both provide 64-bit addressing, and other RISC processor vendors (HP, IBM, and Sun) have announced plans to do that.

On such architectures, 64-bit arithmetic will often be cheaper and faster than 32-bit arithmetic. I believe that designers of NTS might usefully consider the desirability of converting to 64-bit integer arithmetic. Since TEX arithmetic is always done in response to the `\advance`, `\multiply`, and `\divide` operators, or with fixed-point scale factors multiplying dimension registers, the run-time overhead of the change is expected to be minimal, even when it must be simulated in software. This might be most conveniently done by extending dimension fractions to 32 bits, with a 30-bit integer part; counters would be pure 64-bit values.

This larger range of integer and scaled fixed-point arithmetic would drastically reduce the incidence of overflow in intermediate computations, as well as prepare the way for support of much larger address spaces. In TEX, overflow occurs in an operation as simple as converting the small dimension value 3 bp to 3.01125 pt by multiplication by the factor 7227/7200, and because TEX cannot recover from overflow, or sometimes, even detect it, it is difficult to perform even relatively simple scaling operations accurately.

3.9 Absolute page positioning

All computer graphics standards, and PostScript, have included the notion of an absolute page position. TEX regrettably does not, on the grounds that its line-breaking and page-breaking algorithms can result in text tentatively typeset on one page being moved forward or backward a page. In some applications, it is common to require the ability to typeset material at a fixed page location. This is only possible in TEX with code embedded in the output routine, at the point where a page is ready for `\shipout` to the DVI file. Regrettably, such modifications may be infeasible if a complex output routine such as that in LATEX must be dealt with.

NTS should remedy this situation, and I suspect all that is needed is an additional pass over material being typeset. That pass would only be necessary in the event that NTS encountered a command that required absolute page positioning.

3.10 Expression evaluation

TEX's COBOL-like arithmetic is painful to use, and the lack of a general expression parser is a severe deficiency. Although it is possible to write an expression parser in TEX, as Greene has demonstrated [19], I believe that NTS would benefit from a standard facility for the purpose, so that one could write compact and understandable input like this:

```
\count23 = \expr{27 * \count17 -
                 (\count12 + \count13) / 32}
```

The benefits of a powerful expression grammar are widely appreciated by programmers who have been exposed to languages like C and C++, which are particularly rich in this area. NTS designers could usefully study expression grammars of such languages, but would do well in a redesign to reduce their excessive number of operator precedence levels by requiring explicit parentheses to regulate precedence in complex cases.

3.11 Hooks

In the Lisp family of computer languages, the notion of 'hooks' has regularly appeared. In particular, some Lisp dialects have a general mechanism for attaching pieces of code (the 'hooks') to function entry and exit. This is also very easy to do in PostScript. Long experience with TEX, and macro packages like LATEX, has shown that it would be extremely useful to have this facility in NTS. In some cases, it is possible to do this in TEX, with code like

```
\let \oldcode = \code
\def \code
{%
    ...entry hook code...
    \oldcode
    ...exit hook code...
}
```

This is not completely general: there are situations where it fails because TEX expects a particular object to occur next in the input stream (e.g. a box must follow a `\setbox` command). While a

TEX wizard can usually recognize the problem areas, there are enough idiosyncrasies that such useful techniques are infeasible for most users.

In an interpreted language like TEX, commands result in procedure calls inside the interpreter. A proper design of a hook facility could result in an implementation with no overhead at all when hooks were not used, and support for efficient use of multiple entry and exit hooks. I envision a syntax something like this:

```
\addentryhook{\command}{...hook code...}
\addexithook{\command}{...hook code...}
```

Such commands could be repeated, each wrapping one more level of entry and exit hooks around the previous definition of the command.

It is a matter for further study whether additional facilities to strip levels of hooks is desirable. It is possible that grouping would provide sufficient control, since exiting a group would result in loss of non-`\global` hooks added inside the group.

3.12 Low-level input/output primitives

I believe that TEX and its implementation language, Pascal, as well as most older programming languages, have a serious flaw in their I/O model in that they impose an artificial structure that is irrevocably interposed between the program and the file. A superior model is provided in the C programming language, where the fundamental I/O primitives get and put single characters (the smallest directly addressable object on most architectures), and then additional layers of software provide higher-level structuring of files into records, lines, pages, etc.

NTS should provide low-level I/O primitives, perhaps named something like `\getchar` and `\putchar`, that are capable of handling *all* characters representable in the processor.

Existing TEX I/O based on `\read` and `\write` should of course continue to function identically in NTS.

3.13 Input and output translation tables

TEX was developed in an environment where the ASCII character set was widespread. Today, it is in use on a wide variety of computing equipment, and particularly in non-English speaking areas of the world, with a wide variety of character sets. In order to handle character requirements of other languages, ASCII, or ISO8859-1, has been extended with numerous 'code pages', which are essentially specific assignments of characters to the upper 128 of a 256-character table, usually with only minor changes in the lower 128 characters.

Although the future will likely be in 16-bit and 32-bit character sets (notably, ISO 10646M and its Unicode subset), the transition to a universal character set is going to be long and extremely painful. Throughout much of its life, NTS is going to have to deal with this, and I believe that it is essential that the TEX *xchr* and *xord* arrays be accessible via TEX macros. At least one personal computer extension of TEX has already made this possible.

It is essential that these translation tables can be changed on-the-fly during TEX execution, possibly many times, not just at startup time.

3.14 More than 9 parameters

Occasionally, I experience the need for more than 9 parameters in TEX macros. Consider construction of a macro to position four subfigures, each represented by a file name, a width, and a height: 12 parameters are needed. While it is always possible to program around the arbitrary limitation to 9 macro parameters by using tricks like hidden macros that gobble clumps of the arguments, it is decidedly unclean to do so.

If NTS lifts this restriction, then a design decision will have to be made to disambiguate cases where a digit follows a parameter name in a macro body, such as `{#12}`, which in TEX means argument 1 followed by the digit 2, but in NTS, would mean argument 12.

3.15 Arrays

It has occasionally been observed that TEX's `\csname ... \endcsname` provides a way to implement sparse arrays in TEX, much like associative arrays or tables in languages like `awk` [3], `perl`, `icon` [20], and `snobol`. Such techniques rely on the use of efficient name lookup in TEX, which, like the other languages mentioned, uses hashing to achieve $\mathcal{O}(1)$ complexity, independent of table size.

In TEX, one can provide more familiar array syntax while hiding the implementation by using a macro definition like

```
\def \table [#1]%
{%
    \csname \@table [#1]\endcsname
}
```

Unfortunately, this addresses only part of the problem: there is still no way to iterate over the elements of the array, like one can in `awk` with

for (*var* **in** *table*)
{
... loop body...
}

which assigns the next 'subscript' of the array *table* to the variable *var* for the duration of the next iteration of the loop body. The assignments of subscript values happen in no particular order, a consequence of the underlying hash table implementation. In most cases, this does not matter. When a counted loop is required for ascending integer subscripts, a more conventional **for** loop syntax is available.

`awk` also provides a subscript existence test with the syntax

if (*var* **in** *table*)

Extensive experience with associative arrays in those languages that support them has demonstrated their enormous value, particularly in string processing applications, which is partly the domain of TEX and NTS.

It seems to me that such a facility could be added quite cleanly to NTS, and would be found to be of significant value.

3.16 Register limits

Programmers of large TEX macro packages regularly hit the limit of 256 registers (or 16 font families, or 256-character fonts, and maybe even 256 hyphenation tables). These magic numbers are purely a manifestation of an implementation detail in TEX that really should be hidden from the user. NTS should remove all such limits, by allowing them to expand arbitrarily using any integer indexes that are representable in `\count` registers. Ideally, the implementation would use hashing techniques so as to avoid the need for sequential assignment of registers, but since `\newcount`, `\newdimen`, and friends are already in wide use in TEX macro packages, an implementation that allocated table memory assuming all referenced indexes were in use would likely not be overly restrictive, and would allow direct indexing of internal arrays.

TEX currently rejects an attempt to refer to a register (family, font character, ...) number which is out of range, with error messages like

```
\count256
! Bad register code (256).

\char256
! Bad character code (256).
```

so extending the range in NTS should not introduce any compatibility problems.

3.17 String primitives

Although TEX processes text in order to typeset it, it is astonishingly poor in facilities for processing text, and it is surprisingly difficult, and inefficient, to implement such facilities in TEX macros. Even Fortran 77, which predates TEX, has standard functions for searching for a substring in a string, for converting between characters and ordinal values, for concatenation, and for comparison (both in the local character set, and in ASCII). PL/1, which dates from the mid 1960s, has even more string primitives. The widely-used C language offers an even richer set which is clearly documented in an international standard [6] and in a book [34] which contains sample implementations.

NTS would be well advised to incorporate a rich set of string primitives with at least of power and facility of those in the C programming language. Of course, to guarantee identical operation across all platforms, the implementation of NTS would have to provide its own code for each of these string primitives.

3.18 Regular expressions

In the UNIX world, several utilities offer regular-expression pattern matching, and such matching has been a fundamental part of compiler lexical analysis since the 1970s (e.g. see [4] for a detailed theoretical treatment, or UNIX manual pages for such common utilities as `awk`, `ed`, `grep`, and `sed`). Implementations within Emacs-like text editors are available for all major operating systems.

Sadly, the lack of standard library support for this facility led to multiple implementations that differ in small details, and regrettably, the standardization of the C programming language [6] did not remedy that situation. However, POSIX [29, 49] and X/Open [47] have made progress in standardizing the treatment of regular expressions, and extending them to support international character sets. Products of the Free Software Foundation (FSF) provide portable, and well-tested, implementations of such code.

The FSF `regex` package is a highly-optimized implementation that builds on two decades of research in string matching, and it requires several thousand lines of C code. It therefore seems unreasonable, if not infeasible, to implement such a package in TEX macros that would likely take even more code, and be exceedingly slow in execution.

NTS support for a regular-expression pattern-matching library that conforms to international standards would be an extremely valuable extension to TEX's capabilities. Although the implementation of such code is complex, the user interface is surprisingly simple: it can be provided in just four basic primitives in the POSIX specification, and only two of those are required for most applications.

3.19 16-bit and 32-bit character sets and fonts

I commented earlier in Section 3.13 on page 1008 about the 32-bit ISO 10646M character set, and its Unicode 16-bit subset. Already, two operating systems exist which use Unicode: AT&T's Plan 9 (a descendant of UNIX), and Microsoft's Windows NT. Inasmuch as several tens of millions of larger personal computers are potential hosts for Windows NT, its success is a virtual certainty.

It is therefore *imperative* that NTS be able to support at least 16-bit characters, and possibly, 32-bit characters. Plaice [33] has already demonstrated that the TEX source code can be extended to support such large character sets.

Considerable thought will need to be given to the preservation of document portability once much larger character sets come into use.

3.20 Switching between TEX and NTS

Philip Taylor's TUG'93 presentation [40] of the current status of NTS envisaged that the choice between TEX and NTS would be selected only at job startup time, such as by a special command-line option. I believe that such a restriction would be a mistake: even if NTS is successful, there will be a long period in which TEX and NTS must coexist, and consequently, users will find themselves in the position of requiring multiple macro packages, some written in TEX, and some in NTS. It is therefore imperative that it be possible to switch between TEX and NTS on-the-fly.

The major implication of this is that in those few circumstances where the interpretation of a command depends on whether it is NTS or TEX, NTS the program must be able to interpret the command accordingly. This then implies that all user-defined macros, and built-in commands, must internally carry a Boolean tag that identifies which interpretation is required. Of course, if it is possible by careful design to avoid all such conflicts, then the tag will be unnecessary.

3.21 Memory allocation and deallocation

TEX offers no user control over the allocation and deallocation of memory. As in Lisp systems, TEX memory management is automatic. The `\newxxx` commands offer the user a way to create new objects of various types, but no way to subsequently reuse them. With serious limitations on the number of registers, it is common for large macro packages to need a facility for such reuse. Perhaps NTS could provide corresponding `\freexxx` commands to allow such reuse.

Of course, if my suggestions in Section 3.16 on page 1009 are followed, limitations on the number of registers will be removed, and the need for `\freexxx` commands may be reduced.

It may be possible to implement some or all of this feature directly in macros. For example, Spivak provided `\purge` and `\unpurge` in LAMS-TEX [37] to recover memory used by TEX macro packages. The general idea is that a `\purge{foo}` command reads in a file, `foo.tox`, in which every macro from `foo.tex` is `\let` equal to `\undefined`, thereby allowing TEX to reclaim memory. A subsequent `\unpurge{foo}` is equivalent to an `\input` command to reread the original macro package.

3.22 Function `\return`

Both Pascal and TEX lack a **return** statement to exit prematurely from a block of code; the result is that sometimes convoluted programming is needed to avoid executing following code.

It seems to me that NTS would be advised to incorporate a `\return` macro; after all, TEX supports `\endinput` to exit prematurely from the reading of a file. Of course, some programming care would needed for the use of `\return`, in those cases where an apparently-argumentless macro invokes other macros that consume arguments from the input stream; premature return could leave unprocessed arguments in the input. In most cases, however, `\return` should pose no problems.

3.23 Group `\exit`

Just as a `\return` is desirable to exit from the body of a macro, so is a facility for exiting from a group. An `\exitgroup` macro should therefore be a companion addition to NTS.

3.24 Characters with property lists

When a character is typeset on a page, it clearly has certain attributes, or properties. These include page position, font name, font position, color, dimensions, category code, and mode (math, horizontal, vertical, ...). Some of these can be changed, and some cannot (notably, category codes). Of those that can be changed, such as dimensions, the changes may be local to a group, or they may be global. TEX does not supply a uniform method of accessing these properties.

In Lisp systems, objects are created with property lists that can be accessed in a standard manner. By analogy with computer files, one might imagine

that properties might also have access attributes, such as read-only.

It seems to me that NTS might usefully present a property-list model of characters to the macro programmer, and also eliminate the restriction that category codes cannot be changed.

3.25 Dynamic loading and unloading of hyphenation tables

Discussions on TEX-related electronic mail lists have clearly indicated the need to support multiple languages, and therefore hyphenation tables, in some applications. Regrettably, TEX allows these to be loaded only in INITEX. That design mistake should definitely be remedied in NTS. It should be possible to load, and unload, hyphenation tables on-the-fly.

3.26 Memory `\dump/\restore`

INITEX provides a `\dump` command to dump the state of TEX's memory into a special file, called a format (`.fmt`) file, and this can be subsequently loaded by VIRTEX at startup time.

It seems to me that this model is overly restrictive, and that uses could readily be found for a general `\dump/\restore` facility that could be invoked at any time in NTS. Of course, thought would have to be given to exactly what is to be dumped: should it just be macro definitions, or should it also include the contents of registers and the current page box list?

3.27 Font scaling and other font attributes

Vulis' Vector TEX [44] has provided a prototype for how scalable fonts might be supported by NTS, as I believe they must.

On the surface, it would appear that all that is required is an additional `\fontdimen` that would hold a scaling factor for the current font, plus a font property that indicates whether the font is scalable or not. It should be possible to reset it with simple syntax like

```
\fontscale{1200} A
\fontscale{1400} B
\fontscale{1600} C
\fontscale{1000} D
```

Besides *scale*, Vector TEX implements a number of other useful font attributes, including *aspect ratio*, *slant*, *outline*, *shadow*, *fillpattern*, and *smallcaps*. NTS would do well to incorporate such facilities.

3.28 Font rotation

At the time of TEX's birth, phototypesetters were only capable of setting text horizontally or vertically. With the advent of more powerful page description languages, notably, PostScript, this restriction has been eliminated, and text can be typeset along slanted lines, and even along arbitrary curved paths. Although Hoenig has demonstrated [21, 22] that TEX can do this too if it collaborates with METAFONT, it is most certainly not easy to do this.

With PostScript output devices, and suitable DVI driver support, it is possible with TEX at present to employ `\special`s to obtain rotated text.

Rotated characters are needed in many applications, including advertising, and graph labelling. NTS should at the very least provide for the setting of text along lines of arbitrary orientation, so as to standardize the facility.

3.29 Fonts and rules with grey scale, color, and pattern fill

Phototypesetting, and earlier printing technology, did not make it easy to print with more than just black and white. With the advent of color output devices, and page description languages like PostScript to support them, that situation no longer holds.

It should be a fairly simple extension of TEX's rule mechanism to augment *width*, *height*, and *depth* attributes with additional properties, such as grey scale, color, and pattern fill. Almost any commonly-used output device today could easily support grey scale and patterns, and color output capabilities will clearly become widespread in the next few years.

Aside from Vulis' Vector TEX [44] which addresses grey scale and pattern fill, there has been some work already with color in TEX via `\special`s for PostScript DVI drivers [16, 28].

3.30 Line cap styles for rules

TEX's rules are black-filled rectangles with edges aligned with the coordinate axes. PostScript has a `setlinecap` operator [2, p. 506] that allows a choice of butt caps (TEX's choice), round caps, and projecting square caps, each of which has useful applications. Round caps cannot be satisfactorily provided with TEX, because TEX has no filled arc primitive. Projecting square caps can be obtained in TEX by manually adjusting the position and length of rules, but it is decidedly inconvenient to do so.

NTS should provide each of these line cap styles.

3.31 Clipping

Graphics standards and page description languages provide for *clipping*: the removal, or prevention of placement, of material outside a specified region.

While only rectangular clipping regions are supported in most graphics packages, PostScript supports the powerful notion of a clipping path that can be made up of both straight and curved lines, possibly forming disconnected regions. Thus, in PostScript it is possible to use the outline of the letter 'A' to prepare a clipping path over which patterns can be drawn to obtained a pattern-filled letter.

At least for rectangular regions, it would seem straightforward to provide for the setting of text in a rectangular box of specified size, with omission of characters that lie partially, or entirely, outside the box. Suitable commands might be `\cliphbox`, `\clipvbox`, `\clipvtop`, and `\clipvcenter`, or perhaps better, a simple `\clip` command that, issued inside any TEX box primitive, would cause material outside the box to be discarded.

Some thought would need to be given to what should be done with `\llap` and `\rlap` material, which TEX provides to allow material to extend outside its natural enclosing box.

3.32 Job file extension

TEX provides `\jobname` to record the base name of its first input file, but alas, offers no way to override the assumption of a default `.tex` file extension.

In many operating systems, file extensions are used as conventions to indicate something about the contents of files, and other software can take use these to advantage. For example, on UNIX, some compilers will accept file extensions for other languages, and automatically invoke the appropriate language-specific compiler. The `make` utility, possibly the greatest software tool ever written, uses file extensions with file time stamps to identify rules to apply to bring a file up-to-date. Point-and-shoot operations in window systems launch application programs to process the selected file(s), using the file extension (or on the Apple Macintosh, an attribute recorded in the file's resource fork) to select the application program.

I have long followed the convention of using macro-package specific file extensions for TEX files: `.tex` for Plain TEX, `.ltx` for LATEX, `.stx` for SLiTEX, `.atx` for AMS-TEX, and `.lam` for LAMS-TEX. NTS should provide a `\jobextension` or `\jobtype` macro to hold this file extension. It would normally be set by a major macro package, and then used in place of the current hard-coded `.tex` extension by TEX when it needed to read a file for which the extension is omitted. If no such file is found, then NTS should fall back on the old `.tex` default and try it too.

I surmise that addition of this feature to TEX could be done in fewer than ten lines of additional code.

3.33 Usage profiling

Programmers of large applications often want to tune code to increase performance. Some operating systems, notably UNIX, provide several profilers for this purpose. Execution profiles of CPU time per function or subroutine, or per line, and execution counts per line, are essential for reliable tuning.

TEX provides no such facility at present. NTS should at the very least offer a CPU timing facility, perhaps as a simple primitive that returns the CPU execution time since job beginning in suitable small units, either milliseconds, or microseconds. Differencing two samples of a `\cputime` primitive would provide the execution time of a block of code, which is a minimal requirement for measuring code efficiency.

Every computer system has a timer of some sort, and all common operating systems, and some programming languages, make the time accessible to programs, though regrettably without any standard units of measurement or calling interface. I believe that a very small amount of code added to the Web source of NTS, plus small system-dependent alterations in a change file, could provide the needed facility.

TEX provides the `\tracingstats` option to request the dumping of memory usage statistics to the log file. Regrettably, these are not available to the programmer. NTS should supply memory usage statistics as read-only values that can be sampled at any time.

3.34 Word boundary markers

Knuth envisaged that TEX's DVI file output would be used only by DVI drivers to prepare page images. He therefore omitted from the DVI file an important piece of information that is required in other contexts, namely, the marking of word boundaries.

One of the reasons that Berry and coworkers found that the older `troff` was preferable to TEX for multidirectional typesetting and high-quality Arabic ligaturing [11, 38] is the presence of word boundary marks in `troff`'s output.

I believe that NTS should remedy this deficiency.

3.35 Elimination of INITEX

The division of TEX into two separate, but similar, programs, INITEX and VIRTEX, was done because of memory limitations on computers of the time. This has three unfortunate side effects:

- Loading of hyphenation patterns and dumping of format files can only be done in INITEX.
- Novice users are confused by the existence of two separate programs.
- The `trip` torture test is only applied to INITEX, and not to VIRTEX, yet it is the latter that is the program that end users run. I have encountered at least one system on which the `trip` test passed, and two weeks later, a user reported a core dump in (VIR)TEX. Compilers too are programs written by humans, and certainly contain bugs; successful execution of one program does not mean that a related one will be compiled correctly.

I therefore strongly recommend that NTS eliminate the INI and VIR dichotomy, and return to a single executable program that can be validated with a torture test suite. I have been told that TEXtures for the Apple Macintosh, from Blue Sky Research, has already done this.

4 LATEX: the future

Work is well under way for the preparation of a new implementation of LATEX, to be called version 3.0, and documented in several books to be published at the time of its release.

Nevertheless, in conjunction with recommendations for the future of TEX, I thought it would be worthwhile to set down some advice for LATEX development too, even though most of these will not require the new features of NTS to implement.

4.1 LATEX: global vs. local counters and lengths

It is a curious situation that LATEX's `\addtocounter` expands to a `\global \advance`, while the similar `\addtolength` is a bare (local) `\advance`. While the LATEX book [27] does not expose the distinction between global and non-global assignments, the disparity between these two commands can lead to surprises.

It seems to me that both should be local, or both global.

4.2 LATEX: sample input documents

Perhaps the most serious deficiency of the LATEX User's Guide and Reference Manual [27] is that nowhere does it illustrate a minimal input file for LATEX. Consequently, one has to read the book very carefully to divine how to prepare an input file to typeset even a single line of text.

Buerger's book [15], which has received criticism on other grounds, at least presents the user with sample input for a short block of text, a fragment of a scientific article, a recipe, a table, and a few other simple documents.

Borde's books [13, 14] on plain TEX contain nothing but samples of input and output for a wide variety of applications, augmented with discussion of the features of TEX introduced in each example.

The cookbook approach has much to be said for it, and a revised edition of the LATEX User's Guide and Reference Manual would be improved by the introduction of several examples.

4.3 LATEX: minor edit numbers

A great many copies of TEX and LATEX exist on computers around the world. The job of site administrators can be made easier by incorporation of clear edit histories in files, together with an edit number and time stamp that are displayed when programs are run.

Although the LATEX files contain a revision date and major version number, they lack the minor edit number that is traditional in the software industry to characterize the revision history. One has to peruse the `lerrata.tex` file to try to figure out what has changed, but even there, the changes are not date stamped or numbered.

Please, let us have clear revision histories and minor edit numbers in future releases of TEXware.

4.4 LATEX: distinct option and style file extensions

In the current LATEX, minor option and major style file extensions are identical: `.sty`. This makes it impossible for a user to scan a directory of such files and determine which are the major styles, and which are the options.

I urge the LATEX development team to adopt a separate extension, `.opt`, for minor option files, falling back to the old `.sty` extension only if that one fails to identify an existing file.

4.5 LATEX: all options have corresponding file

Some LATEX options do not have corresponding style files, but instead are implemented as macros deeply embedded in major style files. This makes it very difficult to enumerate the document style options that are available. Such enumeration is required for

completion lists in intelligent editors, and for preparation of catalogues of available styles and options.

Please, let us have one file for each minor document style option, even if for certain options, such as point size, it consists of nothing but comments.

4.6 LaTeX: paper names and page dimensions

Although LaTeX markup encourages the author or typist to think in terms of document parts like sections, figures and tables, bibliographies, indexes, and so on, rather than low-level details of the final appearance, the fact is that many, if not most, documents must be prepared with some knowledge of their final form. This is particularly true for wide, or long, tabular material, but is also important when the final touches of elimination of overfull and underfull boxes are being made, through addition of discretionary hyphens, minor rewording, and rarely, alteration of TeX parameters that govern line breaking.

I believe that LaTeX and other TeX-based document formatting systems need to offer style options that correspond to standard paper names, with TeX macros for accessing the paper dimensions. Once the page dimensions are available, it is relatively easy to program styles to make adjustments to `\textheight` and `\textwidth`, and possibly assorted indentation parameters, based on the page dimensions.

My `lptops` utility for turning line printer text files into PostScript recognizes numerous standard paper sizes, and provides easy means to augment the list by local customizations; see Table 1 for the list.

4.7 LaTeX: Font scaling in `picture` mode

When `picture`-mode drawings are resized by adjustment of `\unitlength`, text in the picture does not resize accordingly. If support for scaled fonts is provided in NTS as recommended in Section 3.27 on page 1011, then LaTeX `picture` mode should be enhanced to support them.

4.8 LaTeX: Additional Float placement options

Float placement in LaTeX 2.09 is a perennial problem: figures and tables often float too far away (sometimes dozens of pages) from their point of reference, and there are insufficient placement options.

For the first of these, I believe that the placement algorithm must be revised to ensure that the float never goes more than a page or two away from where it is issued, perhaps by introduction of an exponentially-increasing penalty. There should also be a `\clearfloat` command that can be used to force immediate output of all pending floats. This functionality is clearly already embedded in `\clearpage` and `\cleardoublepage`, but has the nasty side effect of also forcing a page break.

For the second, more precise placement specification should be possible, with these suggested letters to augment the existing `t` (top), `b` (bottom), `h` (here), `p` (separate page containing only floats):

- `x` *exactly* here, even if this results in overfull or underfull pages;
- `e` next even-numbered page, with placement further modified by the existing `t`, `b`, `h`, and `p`;
- `o` next odd-numbered page, with placement further modified by the existing `t`, `b`, `h`, and `p`;
- `f` guaranteed to follow the point of issue, with placement further modified by the existing `t`, `b`, `h`, and `p`.

The last of these is required to cope with style requirements that insist a float may not precede its point of reference; thus, `t` in such a case should select top of the next following page.

4.9 LaTeX: `\path` macro

LaTeX's `\verb` macro for typesetting of short verbatim texts in typewriter font is very convenient. It has one limitation, however: it does not permit a line break in the text.

In computer documentation, it is conventional to represent file names, host names, electronic mail addresses, and sometimes, program variable names, in a typewriter font. Many these tend to be rather long, and it is rather tedious to have to manually insert discretionary hyphens and control sequences for special characters in text like

`Friesland@rz.informatik.uni-hamburg.dbp.de`

Furthermore, discretionary hyphens are undesirable, because it then becomes impossible to tell if a terminal hyphen was part of the original name, or not. Instead, one should use the equivalent of a TeX `\penalty0` to allow a hyphenless line break.

What is needed is a control sequence that works like `\verb`, but allows line breaks at certain user-specifiable characters, without supplying a hyphen. I prepared two prototypes of such a macro, but my TeXpertise was insufficient to produce a truly satisfactory solution, so I posed the problem to a leading TeXpert, Philip Taylor, and he very kindly took up the gauntlet and produced a very workable, and well-documented, implementation of `\path`, which has been freely available since the fall of 1991 in TeX archives in the file `path.sty`. `\path` can be used with *any* TeX-based macro package, not just

Table 1: Common paper names and sizes

Name	Width	Height	Name	Width	Height
A	8.5in	11in	A3L	420mm	297mm
B	11in	17in	A4L	297mm	210mm
C	17in	22in	A5L	210mm	148mm
D	22in	34in	A6L	148mm	105mm
E	34in	44in	A7L	105mm	74mm
Computer-1411	14in	11in	A8L	74mm	52mm
Legal	8.5in	13in	A9L	52mm	37mm
Letter	8.5in	11in	A10L	37mm	26mm
US-Legal	8.5in	14in	B0	1000mm	1414mm
A-L	11in	8.5in	B1	707mm	1000mm
Computer-1411-L	11in	14in	B2	500mm	707mm
Legal-L	13in	8.5in	B3	353mm	500mm
Letter-L	11in	8.5in	B4	250mm	353mm
US-Legal-L	14in	8.5in	B5	176mm	250mm
COM10	4.1in	9.5in	B6	125mm	176mm
DL	110mm	220mm	B0L	1414mm	1000mm
Executive	7.25in	10.5in	B1L	1000mm	707mm
Monarch	3.9in	7.5in	B2L	707mm	500mm
A4Small	210mm	297mm	B3L	500mm	353mm
Ledger	11in	17in	B4L	353mm	250mm
LetterSmall	8.5in	11in	B5L	250mm	176mm
Note	8.5in	11in	B6L	176mm	125mm
A0	841mm	1189mm	C0	1294mm	916mm
A1	594mm	841mm	C1	916mm	647mm
A2	420mm	594mm	C2	647mm	458mm
A3	297mm	420mm	C3	458mm	323mm
A4	210mm	297mm	C4	323mm	229mm
A5	148mm	210mm	C5	229mm	161mm
A6	105mm	148mm	C6	161mm	114mm
A7	74mm	105mm	Octavo	5in	8in
A8	52mm	74mm	Sixmo	6.5in	8in
A9	37mm	52mm	Quarto	8in	10in
A10	26mm	37mm	Foolscap	8.5in	13in
A0L	1189mm	841mm	Government-legal	8.5in	13in
A1L	841mm	594mm	Folio	8.3in	13in
A2L	594mm	420mm			

LaTeX. The characters after which line breaks are permitted can be set by the `\discretionaries` with the same syntax of `\path` and `\verb`; the default setting is equivalent to

```
\discretionaries|~!@$%^&*()_+‘-=#{"}[]:;’<>,.?\/|
```

For electronic mail addresses, a user might elect to change it to

```
\discretionaries+@%!.+
```

Provision is even made for using backslash as an argument delimiter, though I have not yet required this feature.

After two years of frequent use of `\path`, I am convinced of its utility and wide application, and I would very much like to see it incorporated as a standard feature in a new version of LaTeX.

References

[1] Robert A. Adams. Problems on the TeX/PostScript/graphics interface. *TUGBoat*, 11(3):403–408, September 1990.

[2] Adobe Systems Incorporated. *PostScript Language Reference Manual*. Addison-Wesley, Reading, MA, USA, second edition, 1990. ISBN 0-201-18127-4.

[3] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. *The AWK Programming Language*. Addison-Wesley, Reading, MA, USA, 1988. ISBN 0-201-07981-X.

[4] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers—Principles, Techniques,*

and Tools. Addison-Wesley, Reading, MA, USA, 1986. ISBN 0-201-10088-6.

[5] American National Standards Institute, 1430 Broadway, New York, NY 10018, USA. *Military Standard Ada Programming Language*, February 17 1983. Also MIL-STD-1815A.

[6] American National Standards Institute, 1430 Broadway, New York, NY 10018, USA. *American National Standard Programming Language C, ANSI X3.159-1989*, December 14 1989.

[7] American National Standards Institute, 1430 Broadway, New York, NY 10018, USA. *Draft Proposed American National Standard Programming Language Fortran Extended X3.198–199x*, September 24 1990.

[8] Graham Asher. Inside Type & Set. *TUGBoat*, 13(1):13–22, April 1992.

[9] Michael Barr. TEX wish list. *TUGBoat*, 13(2): 223–226, July 1992.

[10] David R. Barstow, Howard E. Shrobe, and Erik Sandewall. *Interactive Programming Environments*. McGraw-Hill, New York, NY, USA, 1984. ISBN 0-07-003885-6. US$34.95.

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

[12] Nelson H. F. Beebe. Comments on the future of TEX and METAFONT. *TUGBoat*, 11(4): 490–494, November 1990.

[13] Arvind Borde. *TEX by Example*. Academic Press, New York, NY, USA, 1992. ISBN 0-12-117650-9.

[14] Arvind Borde. *Mathematical TEX by Example*. Academic Press, New York, NY, USA, 1993. ISBN 0-12-117645-2. xii + 352 pp.

[15] David J. Buerger. *LATEX for Engineers and Scientists*. McGraw-Hill, New York, NY, USA, 1990. ISBN 0-07-008845-4.

[16] Ch. Cérin. Vers la construction de macros de mise en couleur pour TEX. *Cahiers GUTenberg*, 10-11:197–208, septembre 1991.

[17] Malcolm Clark. Changing TEX? *TUGBoat*, 13(2):133–134, July 1992.

[18] William J. Cody, Jr. and William Waite. *Software Manual for the Elementary Functions*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1980. ISBN 0-13-822064-6.

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

[20] Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, second edition, 1990. ISBN 0-13-447889-4. xv + 367 pp.

[21] Alan Hoenig. When TEX and METAFONT talk: Typesetting on curved paths and other special effects. *TUGBoat*, 12(3+4):554–557, November 1991.

[22] Alan Hoenig. When TEX and METAFONT work together. In Zlatuška [48], pages 1–19. ISBN 80-210-0480-0.

[23] *History of Programming Languages*, volume ?? ACM SIGPLAN Notices, 19??

[24] *History of Programming Languages: II*, volume 28. ACM SIGPLAN Notices, March 1993. To be published by ACM Press, 1994.

[25] Donald Knuth and Pierre MacKay. Mixing right-to-left texts with left-to-right texts. *TUGBoat*, 8(1):14, April 1987.

[26] Donald E. Knuth. The future of TEX and METAFONT. *TUGBoat*, 11(4):489, November 1990.

[27] Leslie Lamport. *LATEX—A Document Preparation System—User's Guide and Reference Manual*. Addison-Wesley, Reading, MA, USA, 1985. ISBN 0-201-15790-X.

[28] Daniel Levin. A solution to the color separation problem. *TUGBoat*, 13(2):150–155, July 1992.

[29] Donald A. Lewine. *POSIX programmer's guide: writing portable UNIX programs with the POSIX.1 standard*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 1991. ISBN 0-937175-73-0. LCCN QA76.76.O63 L487 1991b.

[30] Frank Mittelbach. E-TEX: Guidelines for future TEX. *TUGBoat*, 11(3):337–345, September 1990.

[31] IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. IEEE, New York, August 12 1985. A preliminary draft was published in the January 1980 issue of IEEE Computer, together with several companion articles. Available from the IEEE Service Center, Piscataway, NJ, USA.

[32] David A. Patterson and John L. Hennessy. *Computer Architecture—A Quantitative*

Approach. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1990. ISBN 1-55860-069-8.

[33] John Plaice. Language-dependent ligatures. *TUGBoat*, 14(2):xxx, October 1993.

[34] P. J. Plauger. *The Standard C Library*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1992. ISBN 0-13-838012-0.

[35] T. V. Raman. An audio view of (LA)TEX documents. *TUGBoat*, 13(3):372–379, October 1992.

[36] Luigi Semenzato and Edward Wang. A text processing language should be first a programming language. *TUGBoat*, 12(3+4): 434–441, November 1991.

[37] Michael D. Spivak. *LAMS-TEX, The Synthesis*. The TEXplorators Corporation, 3701 W. Alabama, Suite 450-273, Houston, TX 77027, USA, 1990.

[38] Johny Srouji and Daniel Berry. Arabic formatting with `ditroff/ffortid`. *Electronic Publishing—Origination, Dissemination, and Design*, 5(4):163–208, December 1992.

[39] Philip Taylor. TEX: The next generation. *TUGBoat*, 13(2):138, July 1992.

[40] Philip Taylor. Nts: The future of TEX? *TUGBoat*, 14(2):xxx, October 1993.

[41] Phillip Taylor. The future of TEX. In Zlatuška [48], pages 235–254. ISBN 80-210-0480-0.

[42] Michael Vulis. VTEX enhancements to the TEX language. *TUGBoat*, 11(3):429–434, September 1990.

[43] Michael Vulis. Should TEX be extended? *TUGBoat*, 12(3+4):442–447, November 1991.

[44] Michael Vulis. *Modern TEX and its Applications*. CRC Publishers, 2000 Corporate Blvd., Boca Raton, FL 33431, USA, 1992. ISBN 0-8493-4431-X. 275 pp. US$32.95.

[45] Richard Wexelblat (ed.). *History of Programming Languages*. Academic Press, New York, NY, USA, 1983. ISBN 0-12-745040-8. 758 pp. US$61.00.

[46] Niklaus Wirth. *Programming in Modula-2*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, England / etc., second edition, 1983. ISBN 0-387-12206-0.

[47] X/Open Company, Ltd. *X/Open Portability Guide, XSI Commands and Utilities*, volume 1. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1989. ISBN 0-13-685835-X.

[48] Jiří Zlatuška, editor. *EuroTEX '92: Proceedings of the 7th European TEX Conference*. Masarykova University, Prague, Czechoslovakia, 1992. ISBN 80-210-0480-0.

[49] Fred Zlotnick. *The POSIX.1 standard: a programmer's guide*. Benjamin/Cummings Pub. Co., Redwood City, CA, USA, 1991. ISBN 0-8053-9605-5.

◇ Nelson H. F. Beebe
Center for Scientific Computing
Department of Mathematics
University of Utah
Salt Lake City, UT 84112
USA
Tel: +1 801 581 5254
FAX: +1 801 581 4148
Internet: `beebe@math.utah.edu`