Copyright © 1978 American Telephone and Telegraph Company
THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 57, No. 6, July-August 1978
Printed in U. S. A.

UNIX Time-Sharing System:

Document Preparation

By B. W. KERNIGHAN, M. E. LESK, and J. F. OSSANNA, Jr.

(Manuscript received January 6, 1978)

The UNIX operating system provides programs for sophisticated document preparation within the framework of a general-purpose operating system. The document preparation software includes a text editor, programmable text formatters, macro-definition packages for a variety of page layout styles, special processors for mathematical expressions and for tabular material, and numerous supporting programs, such as a spelling-mistake detector. In practice, this collection of facilities has proven to be easy to learn and use, even by secretaries, typists, and other nonspecialists. Experiments have shown that preparation of complicated documents is about twice as fast as on other systems. There are many benefits to using a general-purpose operating system instead of specialized stand-alone terminals or a system dedicated to "word processing." On the UNIX system, these include an excellent software development facility and the ability to share computing and data resources among a community of users.*

I. INTRODUCTION

We use the term *document preparation* to mean the creation, modification, and display of textual material, such as manuals, reports, papers, and books. "Document preparation" seems preferable to "text processing" (which is not particularly precise), or

* UNIX is a trademark of Bell Laboratories.

"word processing" (which has acquired connotations of stand-alone specialized terminals).

Computer-aided document preparation offers some clear benefits. Text need be entered only once. Thereafter, only those portions that need to be changed require editing; the remaining material is left alone. This is a significant labor reduction for any document that must be modified or maintained over a period of time.

There are many other important benefits. Special languages can be used to facilitate the entry of complex material such as tables and mathematical expressions. The style or format of a document can be decoupled from its content; the only format-control information that need be embedded is that describing textual categories and boundaries, such as titles, section headings, paragraphs, and the like. Alternative document styles are then possible through the use of different formatting programs and different interpretations applied to the embedded format control. Furthermore, programs can examine text to detect spelling mistakes, compare versions of documents, and prepare indexes automatically. Machine-generated data can be incorporated in documents; excerpts from documents can be fed to programs without transcription.

A variety of comparatively elegant output devices has become available, supplementing the traditional typewriters, terminals, and line printers; this has led to a much increased interest in automated document preparation. Automated systems are no longer limited to straight text composed in unattractive constant-width characters, but can produce a full range of printed documents in attractive fonts and page layouts. The major example of an output device with significant capabilities is the phototypesetter, which produces very high quality printed output on photographic paper or film. Other devices include typewriter-like terminals capable of high-resolution motion, dot matrix printer-plotters, microfilm recorders, and xerographic printers.

Further advantages accrue when document preparation is done on a general-purpose computer system. One is the opportune sharing of programs and data bases among users; programs originally written for some other purpose may be useful to the document preparer. Having a broad range of users, from typists to scientists, on the same system leads to an unusual degree of cooperation in the preparation of documents.

The UNIX document preparation software includes an easy-to-learn-and-use text editor, **ed**, which is the tool for creating and modifying any kind of text, from documents to data to programs.

Two programmable text formatters, **nroff** and **troff**, provide paginated formatting and allow unusual freedom and flexibility in determining the style of documents. Augmented by various macro-definition packages, **nroff** and **troff** can be programmed to provide footnote processing, multiple-column output, column-length balancing, and automatic figure placement. An equation preprocessor, **eqn**, translates a simple language for describing mathematical expressions into formatter input; a table-construction preprocessor, **tbl**, provides an analogous facility for input of data and text that is to be arranged into tables.

We then mention other programs useful to the document preparer and summarize some comparisons between manual methods of document preparation and methods using UNIX document preparation software.

II. TEXT EDITING

The UNIX text editor **ed** is the basic tool for entering text and for subsequent modifications. We will not try to give a complete description of **ed** here; details may be found in Ref. 1. Rather, we will try to mention those attributes that are most interesting and unusual.

The editor is not specialized to any particular kind of text; it is used for programs, data, and documents alike. It is based on editing commands such as "print" and "substitute," rather than on special function keys, and provides convenient facilities for selecting the text lines to be operated on and altering their contents. Since it does not use special function keys or cursor controls, it does not require a particular kind of input device. Several alternative editors are available that make use of terminals with cursors, but these have been much less widely used; for most purposes, it is fair to say that there is only one editor.

A text editor is often the primary interface between a user and the system, and the program with which most user time is spent. Accordingly, an editor has to be easy to use, and efficient of the user's time—editing commands have to "flow off the fingertips." In accordance with this principle, **ed** is quite terse. Each editor command is a single letter, e.g., **p** for "print," and **d** for "delete." Most commands may be preceded by zero, one, or two "line addresses" to affect, respectively, the "current line" (i.e., the line most recently referenced), the addressed line, or the range of contiguous lines between and including the pair of addresses. There are also shorthands for the current line and the last line of the file. Lines may be addressed by line number, but more common usage is to indicate the position of a line relative to the current or last line. Arithmetic expressions involving line numbers are also permitted:

```
−5,+5p
```

prints from five lines before the current line to five lines after, while

```
$−5,$p
```

prints the last six lines. In both cases, the current line becomes the last line printed, so that subsequent editing operations may begin from there.

Most often, the lines to be affected are specified not by line number, but by "context," that is, by naming some text pattern that occurs in them. The "line address"

```
/abc/
```

refers to the first line after the current line that contains the pattern abc. This line address standing by itself will find and print the next line that contains abc, while

```
/abc/d
```

finds it and deletes it. Context searches begin with the line immediately after the current line, and wrap around from the end of the file to the beginning if necessary. It is also possible to scan the file in the reverse direction by enclosing the pattern in question marks: ?abc? finds the previous abc.

The substitute command s can replace any pattern by any literal string of characters in any group of lines. The command

```
s/ofrmat/format/
```

changes ofrmat to format on the current line, while

```
1,$s/ofrmat/format/
```

changes it everywhere. In both searches and substitute commands, the pattern // is an abbreviation for the most recently used pattern, and & stands for the most recently matched text. Both can be used to avoid repetitive typing. The "undo" command u undoes the most recent substitution.

Text can be added before or after any line, and any group of contiguous lines may be replaced by new lines. "Cut and paste" operations are also possible—any group of lines may be either moved or copied elsewhere. Individual lines may be split or coalesced; text within a line may be rearranged.

The editor does not work on a file directly, but on a copy. Any file may be read into the working text at any point; any contiguous lines may be written out to any file. And any UNIX command may be executed from within the editor, even another instance of the editor.

So far, we have described the basic editor features: this is all that the beginning user needs to know. The editor caters to a wide variety of users, however, and has many features for more sophisticated operations. Patterns are not restricted to literal character strings, but may include several "metacharacters" that specify character classes, repetition of characters or classes, the beginning or end of a line, and so on. For example, the pattern

```
/^[0-9]/
```

searches for the next line that begins with a digit.

Any set of editing commands may be done under control of a "global" command: the editing commands are performed starting at each line that matches a pattern specified in the global command. As the simplest example,

```
g/interesting/p
```

prints all lines that contain `interesting`.

Finally, given the UNIX software for input-output redirection, it is easy to make a "script" of editing commands in advance, then run it on a sequence of files.

The basic pattern-searching and editing capabilities of **ed** have been co-opted into other, more specialized programs as well. The program **grep** ("global regular expression print") prints all input lines that contain a specified pattern; this program is particularly useful for finding the location of an item in a set of files, or for culling items from larger inputs. The program **sed** is a variant of **ed** that performs a set of editing operations on each line of an input stream of arbitrary length.

III. TROFF AND NROFF — BASIC TEXT FORMATTERS

Once a user has entered a document into the file system, it can be formatted and printed by **troff** and **nroff**.[2] These are programmable text formatters that accommodate a wide variety of formatting tasks by providing flexible fundamental tools rather than specific features.

troff supports phototypesetting today on a Graphic Systems phototypesetter and potentially on other typesetters, while nroff produces formatted output for a variety of terminals and line printers, using the full capabilities and resolution of each. troff and nroff are highly compatible with each other, and it is almost always possible to prepare input acceptable to both. Except for device description tables, device-oriented routines, and a relatively small amount of scattered conditionally-compiled code, the source code for these programs is also identical. The device tables permit nroff to understand the entire typesetter character set, printing non-ASCII characters where available or where they can be constructed (by overstriking) on a particular device. The remaining discussion in this section focuses on troff; the behavior of nroff is identical within device capability.

troff is intended to permit unusual freedom in user-designed document styles, while being relatively easy to use for basic formatting tasks. The fundamental operations that troff provides are sufficient for programming complicated formatting tasks. For example, footnote processing, multi-column layout with column balancing, and automatic figure placement are not built-in operations, but are programmed in troff macros when needed. To program in troff, the user writes a set of macro instructions, which expand short abbreviations into the longer command sequences needed for each formatting step. troff may also be instructed to invoke certain macros automatically at particular page positions, such as at the top and near the bottom of the page; other commands are invoked by the user by placing macro calls at paragraphs, section headings, and other relevant boundaries. Once a macro package has been written for some particular style of document, users preparing a document in that style need only provide their text with macro calls at the appropriate points.

The more complex formatting tasks require relatively complex macro packages designed by competent programmers. A well-designed package can be easy to use, and usually permits convenient choice between several related styles. At the simple end of the style spectrum, a newspaper style galley may not require any embedded format control except paragraphing. A simple, paginated style might use only three macros, defining the nature of the top-of-page margin, the bottom-of-page margin, and paragraph breaks.

Input consists of *text* lines, which are destined to be printed, interspersed with *control* lines, which set parameters or otherwise control subsequent processing. Control lines begin with a control character—normally a period—followed by a one- or two-character name that specifies either a built-in *request* or the substitution of a user-defined *macro* in place of the control line. This form is reminiscent of earlier text formatters.[3,4] A typical request is

```
.pl 8.5i
```

which sets the page length to 8.5 inches. Various functions may be introduced anywhere in the input by means of an *escape* character, normally \; examples are \fB, which causes a change to bold font, and \l'3i', which draws a three-inch line.

There are some eighty built-in control-line requests that implement the fundamental operations, allow the setting of parameters, and otherwise affect format control. In addition, some forty escape sequences may appear anywhere to specify certain characters, set indicators, and introduce various functions. Automatic services available include filling and adjusting of text, hyphenation with user control over exceptions, user-settable left, right, and centering tabs, and output line numbering.

User-settable parameters include font, point size, page length, page number, line spacing, line length, indent, and tabs. Functions are available for building brackets, overstriking, drawing vertical and horizontal lines, generating vertical and horizontal motions, and calculating the width of a string. In addition to the parameters that are defined by the formatter, users may define their own parameters, stored in troff variables called "number registers" (for numeric parameters) and "strings" (for character data). These variables may be used in arithmetic and logical expressions to set parameters or to control the invocation of macros or requests.

A *macro* is a user-named set of lines of arbitrary text and format control information. It is interpolated into the input stream either by invoking it by name, or by specifying that it is to be invoked when a particular vertical position on a page is reached. Arguments may be passed to a macro invoked by name. A *string* is a named string of characters that may be interpolated at any point. Macros and strings may be created, redefined, appended to, renamed, and removed. Macros may be nested to an arbitrary depth, limited only by the memory available.

Processed text may be diverted into a macro instead of being output, for footnote collection or to determine the horizontal or vertical size of a block of text before final placement on a page. When reread, diverted text retains its character fonts and sizes and overall dimensions.

A *trap* mechanism provides for action when certain conditions occur. The conditions are position on the current output page, length of a diversion, and an input line count. A macro associated with a vertical page position is automatically invoked when a line of output falls on or after the trap position. For example, reaching a specified place near the bottom of the page could invoke a macro that describes the bottom margin area. Similarly, a vertical position trap may be specified for diverted output. An input line count trap causes a macro to be invoked after reading a specified number of input text lines.

A variety of parameters are available to the user in predefined number registers. In addition, users may define their own registers. Except for certain predefined read-only registers, a number register can be read, written, automatically incremented or decremented, and interpolated into the input in a variety of formats. One common use of user-defined registers is to automatically number sections, paragraphs, lines, etc. A number register may be used any time numerical input is expected or desired. In most circumstances, numerical input may have appended scale factors representing inches, points, ems, etc. Numerical input may be provided by expressions involving a variety of arithmetic and logical operators.

A mechanism is provided for conditionally accepting a group of lines as input. The conditions that may be tested are the value of a numerical expression, the equality of two strings, and the truth of certain built-in conditions.

Certain of the parameters that control text processing constitute an *environment,* which may be switched by the user. It is convenient, for example, to process footnotes in a separate environment from the main text. Environment parameters include line length, line spacing, indent, character size, and the like. In addition, any collected but not yet output lines or words are a part of the environment. Parameters that are global and not switched with the environment include, for example, page length, page position, and macro definitions.

It is not possible to give any substantial examples of troff macro definitions, but we will sketch a few to indicate the general style of use.

The simplest example is to provide pagination—an extra space at the top and bottom of each page. Two macros are usually defined—a *header* macro containing the top-of-page text and spacings, and a *footer* macro containing the bottom-of-page text and spacings. A trap must be placed at vertical position zero to cause the header macro to be invoked and a second trap must be placed at the desired distance from the bottom for the footer. Simple macros merely providing space for the margins could be defined as follows.

```
.de hd          \"  begin header definition
'sp 1i          \"  space 1 inch
..              \"  end of header definition
.de fo          \"  footer
'bp             \"  space to beginning of next page
..              \"  end of footer definition
.wh 0 hd        \"  set trap to invoke hd when at top of page
.wh −1i fo      \"  set trap to invoke fo 1 inch from bottom
```

The sequence \" introduces a troff comment.

The production of multi-column pages requires somewhat more complicated macros. The basic idea is that the header macro records the vertical position of the column tops in a register and initializes a column counter. The footer macro is invoked at the bottom of each column. Normally it increments the column counter, increments the page offset by the column width plus the column separation, and generates a reverse vertical motion to the top of the next column (the place recorded by the header macro). After the last column, however, the page offset is restored and the desired bottom margin functions occur.

Footnote processing is complicated; only the general strategy will be summarized here. A pair of macros is defined that allows the user to indicate the beginning and end of the footnote text. The footnote-start macro begins a diversion that appends to a macro in which footnotes are being collected and changes to the footnote environment. The footnote-end macro terminates the diversion, resets the environment, and moves the footer trap up the page an amount equal to the size of the diverted footnote text. The footer eventually invokes and then removes the macro containing the accumulated footnotes and resets its own trap position. Footnotes that don't fit have their overflow rediverted and are treated as the beginning footnote on the next page.

The use of preprocessors to convert special input languages for equations and tables into troff input means that many documents reach troff containing large amounts of program-generated input. For example, a simple equation might produce dozens of troff input lines and require many string definitions, redefinitions, and detailed numerical computations for proper character positioning. The troff string that finally contains the equation contains many font and size changes and local motion, and so can become very long. All of this demands substantial string storage, efficient storage allocation, larger text buffers than would otherwise be necessary, and the accommodation of large numbers of strings and number registers. Input generated by programs instead of people severely tests program robustness.

IV. MACROS—DECOUPLING CONTENT AND FORMAT

Although troff provides full control over typesetter (or typewriter) features, few users exercise this control directly. Just as programmers have learned to use problem-oriented languages rather than assembly languages, it has proven better for people who prepare documents to describe them in terms of content, rather than specifying point sizes, fonts, etc., in a typesetter-oriented way. This is done by avoiding the detailed commands of troff, and instead embedding in the text only macro commands that expand into troff commands to implement a desired format.

For example, the title of a document might be prefaced by

```
.TL
```

which would expand, for this journal, into "Helvetica Bold font, 14 point type, centered, at top of new page, preceded by copyright notice," but for other journals might be "Times Roman, left adjusted, preceded by a one-inch space," or whatever is desired. In a similar way, there would be macros for other common features of a document, such as author's name, abstract, section, paragraph, and footnote.

Macro packages have been prepared for a variety of document styles. Locally, these include formal and informal internal memoranda; technical reports for external distribution; the Association for Computing Machinery journals; some American Institute of Physics journals; and *The Bell System Technical Journal.* All these macro packages recognize standard macro names for titles, paragraphs, and other document features. Thus, the same input can be made to appear in many different forms, without changing it.

An important advantage of this system is the ease with which new users learn document preparation. It is necessary only to learn the correct way to describe document content and boundaries, not how to control the typesetter at a detailed level. A typist can easily learn the dozen or so most common macros in a few minutes, and another dozen as needed. This entire article uses only about 30 distinct macro calls, rather more than the norm.

Although **nroff** is used for typewriter-like output, and **troff** for photocomposition, they accept exactly the same input language, and thus hide details of particular devices from users. Macro packages also provide a degree of independence: they permit a uniformity of *input*, so that input documents look the same regardless of the output format or device they eventually appear in. This means that to find the title of a document, for example, it is not necessary to know what format is being used to print it. Finally, macros also enforce a uniformity of *output*. Since each output format is defined in appearance by the macro package that generates it, all documents prepared in that format will look the same.

V. EQN—A PREPROCESSOR FOR MATHEMATICAL EXPRESSIONS

Much of the work of Bell Laboratories is described in technical reports and papers containing significant amounts of mathematics. Mathematical material is difficult to type and expensive to typeset by traditional methods. Because of positioning requirements and the multiplicity of characters, sizes, and fonts, it is not feasible for a human to typeset mathematics directly with **troff** commands. **troff** is richly endowed with the facilities needed for preparing mathematical expressions, such as arbitrary horizontal and vertical motions, line-drawing, size changing, etc., but it is not easy to use these facilities directly because of the difficulty of deciding the degree of size change and motion suitable in every circumstance. For this reason, a language for describing mathematical expressions was designed; this language is translated into **troff** by a program called **eqn**.

An important requirement is that the language should be easy to learn and use by people who don't know mathematics, computing, or typesetting. This implies that normal mathematical conventions about operator precedence, parentheses, and the like cannot be used, for otherwise the user would have to understand what was being typed. Further, there should be very few rules, keywords, special symbols, and few exceptions to the rules. Finally, standard actions should take place automatically—size and font changes should follow normal mathematical usage without user intervention.

When a document is typed, mathematical expressions are entered as part of the text, but are marked by user-settable delimiters. **eqn** reads this input and passes through untouched those parts that are not mathematics. At the same time, it converts the mathematical parts into the necessary troff commands. Thus normal usage is a pipeline of the form

```
eqn files | troff
```

The language is defined by a Yacc[5] grammar to insure regularity and ease of change. We will not describe the eqn language in detail; see Refs. 6 and 7. Nonetheless, it is worth showing a few examples to give a feeling for the language. Throughout this section we write expressions exactly as they are typed by the user, except that we omit the delimiters that mark the beginning and end of each expression.

eqn is an oral (or perhaps aural) language. To produce

$$2\pi \int \sin(\omega t)\, dt$$

one writes

```
2 pi int sin ( omega t)dt
```

Each "word" in the input is looked up in a table. In this case, pi and omega are recognized as Greek letters, int is a special character, and sin is to be placed in Roman font instead of italic, following conventional practice. Parentheses and digits are also made Roman, and spacing is adjusted around characters to give a more pleasing appearance.

Subscripts, superscripts, fractions, radicals, and the like are introduced by words used as operators:

$$\frac{x^2}{a^2} = \sqrt{pz^2+qz+r}$$

is produced by

```
x sup 2 over a sup 2 ~=~ sqrt {pz sup 2 + qz + r}
```

The operator sub produces a subscript in the same manner as sup produces a superscript. Braces { and } are used to group items that are to be treated as a unit, such as all the terms to go under the radical. eqn input is free-form, so blanks and new lines can be used freely to make the input easier to type, read, and subsequently edit. The tilde ~ is used to force extra space into the output when needed.

More complicated expressions are built from the same piece parts, and perhaps a few new ones. For example,

```
erf (z) ~=~ 2 over sqrt pi int sub 0 sup z
e sup -t sup 2 dt
```

produces

$$erf(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} dt$$

while

```
zeta (s) ~=~ sum from k=1 to inf
k sup -s ~~~ ( Re ~ s > 1)
```

is

$$\zeta(s) = \sum_{k=1}^{\infty} k^{-s} \quad (\mathrm{Re}\ s>1)$$

and

```
lim from {x -> pi /2} ( tan ~x) sup {sin~2x} ~=~ 1
```

yields

$$\lim_{x \to \pi/2} (\tan x)^{\sin 2x} = 1$$

In addition, there are built-up brackets, braces, etc.; matrices; diacritical marks such as dots and bars; font and size changes to override defaults; facilities for lining up equations; and macro substitution.

Because not all potential users have access to a typesetter, there is also a compatible version of eqn that interfaces to nroff for producing output on terminals capable of half-line motions and printing special characters. The quality of terminal output leaves something to be desired, but it is often adequate for proofreading and some internal uses.

The eqn language has proven to be easy to learn and use; at the present time, well over a hundred typists and secretaries use it at Bell Laboratories. Most are either self-taught, or have learned it as part of a course in UNIX system procedures taught by other secretaries and typists. Empirically, mathematically trained users (mathematicians, physicists, etc.) can learn enough eqn in a few minutes to begin useful work, for its syntax and rules are very similar to the way that mathematics is actually spoken. Persons not trained in mathematics take longer to get started, because the language is less familiar, but it is still true that an hour or two of instruction is enough to begin doing useful work.

By intent, eqn does not know very much about typesetting; in general, it lets troff do as much of the job as possible, including all character-width computations. In this way, eqn can be relatively independent of the particular character set, and even of the typesetter being used.

The basic design decision to make a separate language and program distinct from troff does have some drawbacks, because it is not easy for eqn to make a decision based on the way that troff will produce the output. The programs are very loosely coupled. Nonetheless, these drawbacks seem unimportant compared to the benefits of having a language that is easily mastered, and a program that is separate from the main typesetting program. Changes in one program generally do not affect the other; both programs are smaller than they would be if they were combined. And, of course if one doesn't use eqn, there is no cost, since troff doesn't contain any code for it.

VI. TBL—A PREPROCESSOR FOR TABLES

Tables also present typographic difficulties. The primary difficulty is deciding where columns should be placed to accommodate the range of widths of the various table entries. It is even harder to arrange for various lines or boxes to be drawn within the table in a suitable way. tbl[8] is a table construction program that is also an independent preprocessor, quite analogous to eqn.

tbl simplifies entering tabular data, which may be tedious to type or may be generated by a program, by separating the table format from its contents. Each table specification contains three parts: a set of global options affecting the whole table, such as "center" or "box"; then a set of commands describing the format of each line of the table; and finally the table data. Each specification describes the alignment of the fields on a line, so that the description

```
L R R
```

indicates a line with three fields, one left adjusted and two right adjusted. Other kinds of fields are "C" (centered) and "N" (numerical adjustment), with "S" (spanned) used to continue a field across more than one column. For example,

```
C S S
L N N
```

describes a table whose first line is a centered heading spanning three columns; the three columns are left-adjusted, numerically adjusted, and numerically adjusted respectively. If there are more lines of data than of specifications (the normal case), the last specification applies to all remaining data lines.

A sample table in the format above might be

Position of Major Cities		
Tokyo	35°45′ N	139°46′ E
New York	40°43′ N	74°01′ W
London	51°30′ N	0°10′ W
Singapore	1°17′ N	103°51′ E

The input to produce the above table, with tab characters shown by the symbol Ⓣ, is as follows:

```
.TS
center, box;
C S S
L N N.
Position of Major Cities
TokyoⓉ35°45′ NⓉ139°46′ E
New YorkⓉ40°43′ NⓉ74°01′ W
LondonⓉ51°30′ NⓉ0°10′ W
SingaporeⓉ1°17′ NⓉ103°51′ E
.TE
```

tbl also provides facilities for including blocks of text within a table. A block of text may contain any normal typesetting commands, and may be adjusted and filled as usual. tbl will arrange for adequate space to be left for it and will position it correctly. For example, the table on the next page uses text blocks, line and box drawing, size and font changes, and the facility for centering vertical placement of the headings (compare the heading of column 3 with that of columns 1 and 2). Note that there is no difficulty with equations in tables. In fact, there is sometimes a choice between writing a matrix with the matrix commands of eqn or making a table of equations. Typically, the typist picks whichever program is more familiar.

The tbl program writes troff code as output, just as eqn does. This code computes the width of each table entry, decides where to place the columns and lines separating them, and prints the table. tbl itself does not understand typesetting: it does not know the

Functional Systems		
Function Number	Function Type	Solution
1	LINEAR	Systems of equations all of which are linear can be solved by Gaussian elimination.
2	POLYNOMIAL	Depending on the initial guess, Newton's method ($f_{i+1}=f_i-\frac{f_i'}{f_i}$) will often converge on such systems.
3	ALGEBRAIC	The program ZONE by J. L. Blue will solve systems for which an accurate initial guess is not known.

widths of characters, and may (in the case of equations in tables) have no knowledge of the height, either. However, it writes **troff** output that computes these sizes, and adjusts the table accordingly. Thus tables can be printed on any device and in any font without additional work.

Most of the comments about using **eqn** apply to **tbl** as well: it is easy to learn and is in wide use at Bell Laboratories. Since it is a program separate from **troff**, it need not be learned, used, or paid for if no tables are present. Comparatively few users need to know all of the tools: typically, the workload in one area may be mathematical, in another area statistical and tabular, and in another only ordinary text.

VII. OTHER SUPPORTING SOFTWARE

One advantage of doing document preparation in a general-purpose computing environment instead of with a specialized word processing system is that programs not directly related to document preparation may often be used to make the job easier. In this section, we discuss some examples from our experience.

One of the most tedious tasks in document preparation is detection of spelling and typographical errors. Existing data bases originally obtained for other purposes are used by a program called **spell**, which detects potential spelling mistakes. Machine-readable dictionaries (more precisely, word lists) have been available for some time. Ours was originally used for testing hyphenation algorithms and for checking voice synthesizer programs. It was realized, however, that a rudimentary program for detecting spelling mistakes could be made simply by comparing each word in a document with each word in the dictionary; any word in the document but not in the dictionary is a potential misspelling.

The first program for this approach was developed in a few minutes by combining existing UNIX utilities for sorting, comparing, etc. This was sufficiently promising that additional small programs were written to handle inflected forms like plurals and past participles. The resulting program was quite useful, for it provided a good match of computer capabilities to human ones. The machine can reduce a very large document to a tractable list of suspicious words that a human can rapidly scan to detect the genuine errors.

Naturally, normal output from **spell** contains not only legitimate errors, but a fair amount of technical jargon and some proper names. The next step is to use that output to refine the dictionary. In fact, we have carried this step to its logical conclusion, by creating a brand new dictionary that contains only words culled from documents. This new dictionary is about one-third the size of the original, and produces rather better results.

One of the more interesting peripheral devices supported by the UNIX system is an inexpensive voice synthesizer.[9] The program **speak**[10] uses this synthesizer to pronounce arbitrary text. Speaking text has proven especially handy for proofreading tedious data like lists of numbers: the machine speaks the numbers, while a person reads a list in parallel.

Another example of a borrowed program is **diff**,[11] which compares two inputs and prepares a list of all the places in which they differ. Normally, **diff** is used for comparing two versions of a program, as a check on the changes that have been made. But of course it can also be used on two versions of a document as well. In fact, the **diff** output can be captured and used to produce a set of **troff** commands that will print the new version with marginal bars indicating the places where the document has been changed.

We have already mentioned two major preprocessors for **troff** and **nroff**, for mathematics and tables. The same approach, of writing a separate program instead of cluttering up an existing one, has been applied to *post*processors as well. Typically, these postprocessors are concerned with matching **troff** or **nroff** output with the characteristics of some different output device. One example is a processor called **col** that converts **nroff** output containing reverse motions (e.g., multi-column output) into page images suitable for printing on devices incapable of reverse motion. Another example is a program that converts **troff** output intended for a phototypesetter into a form suitable for display on the screen of a Tektronix 4014 terminal (or analogous graphic devices). This permits a view of the formatted document without actually printing it; this is especially convenient for checking page layout.

One final area worth mentioning concerns the problem of training new users. Since there seems to be no substitute for hands-on experience, a program called **learn** was written to walk new users through sets of lessons.[12] Lesson scripts are available for fundamentals of UNIX file handling commands, the editor **ed**, and **eqn**, as well as for topics not related to document preparation. **learn** has been heavily used in the courses taught by secretaries and typists for their colleagues.

VIII. EXPERIENCE

UNIX document preparation software has now been used for several years within Bell Laboratories, with many secretaries and typists in technical organizations routinely preparing technical memoranda and papers. Several books[13-19] printed with this software have been published directly from camera-ready copy. Technical articles have been prepared in camera-ready form for periodicals ranging from the *Journal of the ACM* to *Science*.

The longest-running use of the UNIX system for document preparation is in the Bell Laboratories Legal and Patent Division, where patent applications have been prepared on a UNIX system for nearly seven years. Computer program documentation has been produced for several years by clerks using UNIX facilities at the Business Information Systems Programs Area of Bell Laboratories. More recently, the "word processing" centers at Bell Laboratories have begun significant use of the UNIX system because of its ability to handle complicated material effectively.

It can be difficult to evaluate the cost-effectiveness of computer-aided versus manual documentation preparation. We took advantage of the interest of the American Physical Society in the UNIX system to make a systematic comparison of costs of their traditional typewriter composition and a UNIX document preparation system. Five manuscripts submitted to *Physical Review Letters* were typeset at Bell Laboratories, using the programs described above to handle the text, equations, tables, and special layout of the journal.

On the basis of these experiments, it appears that computerized typesetting of difficult material is substantially cheaper than typewriter composition. The primary cost of page composition is keyboarding, and the aids provided by UNIX software to facilitate input of complex mathematical and tabular material reduce input time significantly. Typing and correcting articles on the UNIX system, with an experienced typist, was between 1.5 and 3.3 times as fast as typewriter composition. Over the trial set of manuscripts, input using the UNIX system was 2.4 times as fast. These documents were extremely complicated, with many difficult equations. Typists at *Physical Review Letters* averaged less than four pages per day; whereas our (admittedly very proficient) UNIX system typist could type a page in 30 minutes. We estimate a very substantial saving in production cost for camera-ready pages using a UNIX system instead of conventional composition or typewriting. A typical UNIX system for photocomposition of *Physical Review* style pages might produce 200 finished pages per day on a capital investment of about $200,000 and with 20 typists.

The advantage of the UNIX system is greatest when mathematics and tables abound in a document. For example, it is a great time saving that keys need never be changed because all equation input is ordinary text. The automatic page layout saves time when multiple drafts, versions, or editions of a document are needed. Further details of this comparison can be found in Ref. 20.

IX. CONCLUSIONS

It is important to note that these document preparation programs are simply application programs running on a general-purpose system. Any document preparation user can exercise any command whenever desired.

As mentioned above, a surprising number of the programming utilities are directly or indirectly useful in document preparation. For example, the program that makes cross-reference listings of computer programs is largely identical with the one that makes keyword-in-context indexes of natural language text. It is also easy to use the programming facilities to generate small utilities, such as one which checks the consistency of equation usage.

Besides applying programming utilities to text processing, we also apply document processors to programs and numerical data. Statistical data are often extracted from program output and inserted into documents. Computer programs are often printed in papers and books; because the programs are tested and typeset from the same source file, transcription errors are eliminated.

In addition to the technical advantages of having programming and word processing on the same machine, there can be personnel advantages. The fact that secretaries and typists work on the same system as the authors allows both to share the document preparation job. A document may be typed originally by a secretary, with the author doing the corrections; in the case of an author who types rough drafts but doesn't like editing after proofreading, the reverse may occur. We have observed the full spectrum, from authors who give hand-written material to typists in the traditional manner to those who compose at the terminal and do their own typesetting. Most authors, however, seem to operate somewhere in between.

The UNIX system provides a convenient and cost-effective environment for document preparation. A first-class program development facility encourages the development of good tools. The ability to use preprocessors has enabled us to write separate languages for mathematics, tables, and several other formatting tasks. The separate programs are easier to learn than if they were all jammed into one package, and are vastly easier to maintain as well. And since all of this takes place within a general-purpose operating system, programs and data can be used as convenient, whether they are intended for document preparation or not.

REFERENCES

1. K. Thompson and D. M. Ritchie, *UNIX Programmer's Manual,* Bell Laboratories, May 1975. See ED (I).
2. J. F. Ossanna, "NROFF/TROFF User's Manual," Comp. Sci. Tech. Rep. No. 54, Bell Laboratories (April 1977).
3. J. E. Saltzer, "Runoff," in *The Compatible Time-Sharing System,* ed. P. A. Crisman, Cambridge, Mass.: M.I.T. Press (1965).
4. M. D. McIlroy, "The Roff Text Formatter," Computer Center Report MHCC-005, Bell Laboratories (October 1972).
5. S. C. Johnson, "Yacc — Yet Another Compiler-Compiler," Comp. Sci. Tech. Rep. No. 32, Bell Laboratories (July 1975).
6. B. W. Kernighan and L. L. Cherry, "A System for Typesetting Mathematics," Commun. Assn. Comp. Mach., *18* (March 1975), pp. 151-157.
7. B. W. Kernighan and L. L. Cherry, "A System for Typesetting Mathematics," Comp. Sci. Tech. Rep. No. 17, Bell Laboratories (April 1977).
8. M. E. Lesk, "Tbl — A Program to Format Tables," Comp. Sci. Tech. Rep. No. 49, Bell Laboratories (September 1976).
9. Federal Screw Works, *Votrax ML-1 Multi-Lingual Voice System.*
10. M. D. McIlroy, "Synthetic English Speech by Rule," Comp. Sci. Tech. Rep. No. 14, Bell Laboratories (March 1974).
11. J. W. Hunt and M. D. McIlroy, "An Algorithm for Differential File Comparison," Comp. Sci. Tech. Rep. No. 41, Bell Laboratories (June 1976).
12. B. W. Kernighan and M. E. Lesk, unpublished work (1976).
13. B. W. Kernighan and P. J. Plauger, *The Elements of Programming Style,* New York: McGraw-Hill, 1974.
14. C. H. Sequin and M. F. Tompsett, *Charge Transfer Devices,* New York: Academic Press, 1975.
15. B. W. Kernighan and P. J. Plauger, *Software Tools,* Reading, Mass.: Addison-Wesley, 1976.

16. T. A. Dolotta et al., *Data Processing in 1980-1985: A Study of Potential Limitations to Progress,* New York: Wiley-Interscience, 1976.
17. A. V. Aho and J. D. Ullman, *Principles of Compiler Design,* Reading, Mass.: Addison-Wesley, 1977.
18. Committee on Impacts of Stratospheric Change, *Halocarbons: Environmental Effects of Chlorofluoromethane Release,* Washington, D. C.: National Academy of Sciences, 1977.
19. W. H. Williams, *A Sampler on Sampling,* New York: John Wiley & Sons, 1977.
20. M. E. Lesk and B. W. Kernighan, "Computer Typesetting of Technical Journals on UNIX," Proc. AFIPS NCC, *46* (1977), pp. 879-888.